JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

A CATEGORICAL FRAMEWORK OF CONSISTENCY IN CURRENT SYSTEMS

Mr Bani Bhusan Praharaj [1]

Abstract:

A current system involves several executing components. Such a system usually allowsto carry out multiple tasks simultaneously, which can speed up the computational work of software substantially. To develop concurrent systems, *process-oriented programming* is considered naturally fit the design and implementation [1]. This kind of programming is founded on *process algebra* [2], Hoare's Communicating Sequential Processes (CSP) [3, 4, 5] and Milner's π-Calculus [6], which consider a concurrent system as a set of interacting *processes* with *messages* passing through *chan- nels* [1, 7]. It has been considered that process-oriented design and implementation could provide systems with known safety properties to prevent *deadlock*, *livelock*, *process starvation* [1]. Con- current systems developed by process-oriented approach are able to be efficiently distributed across multiple processors and clusters of machines [7].

Chapter 1

Introduction

A current system involves several executing components. Such a system usually allowsto carry out multiple tasks simultaneously, which can speed up the computational work of software substantially. To develop concurrent systems, *process-oriented programming* is considered naturally fit the design and implementation [1]. This kind of programming is founded on *process algebra* [2], Hoare's Communicating Sequential Processes (CSP) [3, 4, 5] and Milner's π-Calculus [6], which consider a concurrent system as a set of interacting *processes* with *messages* passing through *chan- nels* [1, 7]. It has been considered that process-oriented design and implementation could provide systems with known safety properties to prevent *deadlock*, *livelock*, *process starvation* [1]. Con- current systems developed by process-oriented approach are able to be efficiently distributed across multiple processors and clusters of machines [7].

However, design and implementation are usually at different levels of abstraction in software development process [8]. It is challenging to incorporate knowledge and experience to manage the consistency between these phases in developing concurrent systems [8]. Especially, when many processes communicate simultaneously, a concurrent system may exhibit a large number of different behaviors. Inconsistencies arising would bring errors to the production of concurrent systems [9], which would prove fatal to the systems in areas with no-tolerance for failure. To deal with such a challenge, verification plays a crucial role in reducing, or even preventing, the introduction of errors in design and implementation of a concurrent system [10]. There has been much research

in verifying consistency between design and implementation. However, most of the existing re- search [10, 11, 12, 13] has been carried out is not targeted for concurrent systems developed in process-oriented languages. Specifically, we currently lack formal analysis techniques to analyze consistency of communications between design and implementation of concurrent systems devel- oped in process-oriented languages.

Inspired by Hoare's vision of category and functor as tools to formalizing relationships be- tween design, correctness proof, and programming languages [14], our research is built upon the research [15] which has obtained results that has validated the vision. As a continuation of re- search [16], The aim of this research is to provide a novel categorical framework to formally verify consistency of communications between process-oriented design and implementation of concurrent systems.

This chapter gives an overview of the structure of the thesis. Section 1.1 gives a short introduc- tion to the aspects that motivated our research. Section 1.2 describes the research problems we are interested in. In section 1.3, we propose our research goal and objectives. Section 1.4 provides the thesis organization.

Motivation

In this section, several aspects that motivated this work and possess the potential to be researched upon are highlighted.

Importance of Concurrent Systems

In the real world, many things happen at the same time. As a software system needs to model the part of the world for which it is to be used, naturally concurrency fits in the software systems. It consists of simultaneously executing components, which provide the ability to do more than one task at a time. By performing multiple tasks concurrently, computational work of software could be speeded up substantially [17]. With the continuous development of hardware and software, concurrent systems widely apply to a range of areas.

Advantages of Process-Oriented Programming

Process-oriented programming languages are naturally suited to the development of concurrent systems [7]. These kinds of programming languages usually have several concurrent processes interacting through message-passing over channels [1]. A process encapsulates a collection of data and methods for managing that data. Data and methods inside the process cannot be manipulated outside processes [1]. External processes only can pass messages through channels to the process for using the data and methods [1].

It is considered that process-oriented programming languages satisfy several requirements, such as *safe concurrency*, *scalability*, *evolvability*, and *weak coupling between components* [18]. A process-oriented software is constructed as a network of isolated concurrent processes that inter- act only using channels [7]. With mechanisms drawn from CSP and π-Calculus, design rules are able to provide systems with known safety properties [7] to prevent deadlock, livelock, process starvation [1]. Since process-oriented programs expose by their nature a high degree of explicit con- currency, they can be efficiently distributed across multiple processors and clusters of machines [7].

Importance of Verifying Consistency between Design and Implementation of Concurrent Systems

In software development process, design and implementation are at different levels of abstrac- tion [19]. Incorporating knowledge and experience to manage design and implementation of con- current systems is considered a serious challenge [8]. Inconsistencies arising would introduce errors to the production of concurrent systems [19], which would be fatal to the systems in areas with zero tolerance for failure.

As a concurrent system would exhibit different behaviors, testing concurrent systems has a limited role due to the difficulties of making tests to cover all the possible executions [9, 10]. To manage such challenges, verification techniques are necessary for proving the consistency between design and implementation of concurrent systems [11]. Among several verification techniques, deductive verification and model checking are widely considered and adopted [9, 10]. However, deductive verification requires insight as well as significant mathematical calculation, and model checking experiences a major obstacle called *state-space explosion* [9, 10].

The above motivated the work presented in this thesis, aimed at solving the research problem stated in the following section.

Problem Statement

For concurrent systems developed by process-oriented programming languages, this research focuses on verifying consistency between design and implementation. We propose a category theory approach to model concurrent systems with the purpose of exploring answers for the following research questions:

- **RQ1**. How do we model communications between processes in design of concurrent systems with category theory?
- **RQ2**. How do we model communications between processes in implementation of concurrent systems with category theory?
- **RQ3**. How can category theory be used to determine whether or not the implementation is consistent with the designed communications of concurrent processes?

Research Goal and Objectives

To solve the research problems, our goal is to build the categorical framework for process- oriented languages (see Fig. 1.1). This framework can be used to verify the consistency of process communications between design and implementation. In this framework, we propose transforma- tion between the formalisms selected to model design and implementation of concurrent systems into categorical models.

To build the framework, we have the following objectives:

- **OBJ1**: model and analyze process communications in design of concurrent systems with CSP.
- **OBJ2**: implement the concurrent systems in Erasmus by refining the design.

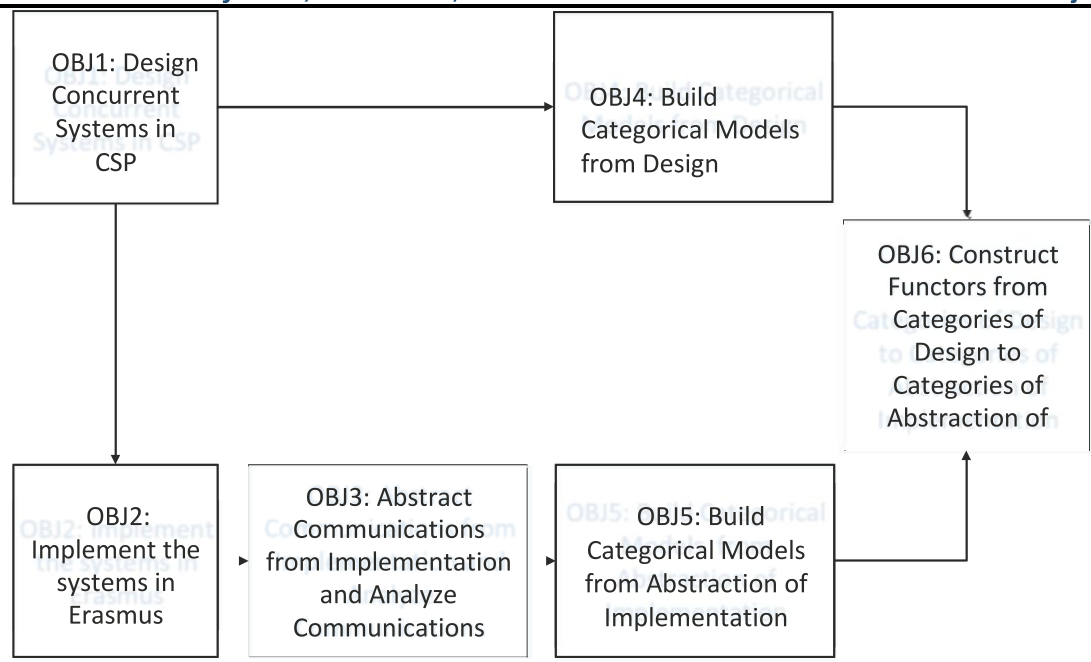

Figure 1.1: Research Goal and Objectives

- **OBJ3**: abstract and analyze process communications from implementation in Erasmus.
- **OBJ4**: define structural transformations from design to categorical models of design.
- **OBJ5**: define structural transformations from abstraction of implementation to categorical models of implementation.
- **OBJ6**: verify consistency of process communications between categorical models of design and implementation.

Specifically, OBJ1 and OBJ4 aim to answer research question RQ1, OBJ2, OBJ3 and OBJ5 aim to answer research question RQ2, and OBJ6 aim to answer research question RQ3.

Thesis Organization

The thesis is structured as follows: Chapter 2 reviews the theoretical background of the research. Chapter 3 presents our innovative categorical framework for verification. Chapter 4 introduces how to use the framework to verify process communications with traces in a running example. Chapter 5 introduces using the framework to verify process communications with failures in a running exam- ple with three implementation scenarios. Chapter 6 provides algorithms for automatically generat- ing failures of process communications, and constructing categories and functors for verification. Chapter 7 shows how to use data flow and category theory to verify process communications in the implementation against properties of process communications in Erasmus. Finally, in Chapter 8, thesis conclusions and possible future works are provided.

Chapter 2

Background and Related Work

This chapter introduces the theoretical background and related work of the research. Section 2.1 presents an overview of concurrent systems. Section 2.2 explains process-oriented languages. Sec- tion 2.3 presents communicating sequential processes (CSP). Section 2.4 introduces necessary in- formation of Erasmus. Section 2.5 briefs techniques used in verification. Section 2.6 provides the basic definitions and terminologies for the Galois connection in abstract interpretation. Section 2.7 introduces basics of data flow analysis. In Section 2.8, category theory and some definitions are explained.

Concurrent Systems

With the increase in demand of processing multiple tasks simultaneously and the prevalence of parallel computer hardware, concurrency has been at the center of software engineering since its inception [7]. Usually, a concurrent system consists of a set of processes that can execute and communicate with each other. However, this can lead to a combinatorial explosion of possible execution as well as that of communications between processes [10].

To build a concurrent system, conventional programming languages need to make some adap- tations [18]. In object-oriented programming languages, one such adaptation is the use of threads to handle concurrency. Nevertheless, different programming styles of threads would make the soft- ware behave uncertainties [20]. Thus, more complexity is added to object-oriented programming languages that are already very complex [18].

Process-Oriented Languages

Process-oriented languages are considered to be the next programming paradigm that naturally fit the development of concurrent systems [1, 7]. Many process-oriented programming languages are based on process algebra CSP and π-Calculus. Process-oriented programming is based on pro- cesses that communicate by passing messages through channels rather than objects invoking one another's methods in object-oriented programming [7, 18]. It is considered that process-oriented languages satisfy several requirements, such as safe concurrency, scalability, evolvability, modeling capabilities and modularity, and weak coupling between components [21].

To support process-oriented programming, several languages and libraries are designed, such as *Erasmus* [22], *occam-π* [23] and *JCSP* [24]. Though JCSP provides processes and channels for computation and passing messages, it is still a library added to object-oriented programming language Java. Occam-*π* programs are constructed as process networks with processes as nodes and channels as edges for passing messages. A channel is typed to specify the kinds of messages that can be passed through itself, while a protocol is defined to specify a sequence of messages that can be passed through a channel. Besides, in occam-*π*, a lower-level process network can be abstracted as a node in a higher-level process network, which conforms to the software engineering principle: separation of concerns. Compared with occam-*π*, Erasmus has similar features. In Erasmus, a port that is of the type of a protocol works as an interface of a process to connect to a channel. A process can have several ports. Each port of a process specifies the types and the sequences of messages that the port receives or sends through a channel. With the notion of port, it helps to specify and analyze passing messages between processes and channels. Moreover, some features of Erasmus can be modified and adapted based on the needs of our research when doing the categorical analysis. In this research, Erasmus is chosen to implement concurrent systems.

Communicating Sequential Processes (CSP)

CSP was first proposed by Hoare as a language in 1978 [3], then was refined toward specification- oriented with its process algebraic form in 1985 [4], and has evolved later by Roscoe around 2010 [5]. It has been widely used to specify, design and implement concurrent systems. CSP specifies and models processes in a concurrent system that communicate with their external environment. The construction of a process depends on a set of all *events* that occur on the process. This set of all events is called an *alphabet*. A process in CSP can be described by a set of *traces*. Each trace is a sequence of events. Trace can be extended to *failure* and *divergence* in order to describe safety and liveness of the process. In CSP, a process is defined as (*alphabet*, *failures*, *divergences*) [4, 5], which will be explained in Chapter 3. If a process is assumed not to become *chaos*, (*alphabet*, *failures*) is enough to describe safety and liveness of the process [1]. Processes can be assembled together as a system, where they can interact with each other and with their external environment. Such interac- tions are called *communications*, which are synchronized. If one process needs to communicate to another process, a channel is required between them to receive the input of messages and pass the output of them at the same time. Also, several operators are defined to describe the relationships between processes. Given two processes P and Q, CSP can calculate sequence P ; Q, deterministic choice P Q Q, non-deterministic choice P H Q, parallel execution $P \mid Q$, and iteration, using the recursion operator $\mu P : A \cdot F(P)$.

Erasmus

Erasmus is one of process-oriented programming languages, which is based on the idea of CSP but with some differences [18, 21, 22, 25]. An Erasmus program consists of *cells*, *processes*, *ports*, *protocols* and *channels* . A cell, containing a collection of one or more processes or cells, provides the structuring mechanism for an Erasmus program. A process is a self-contained entity which performs computations, and communicates with other processes through its ports. A port, which is of a type of protocol, usually serves as an interface of a process for sending and receiving messages. A protocol specifies the type and the orderings of messages that can be sent and received by ports of the type of this protocol. A channel, which is of a type of protocol, must be built between two ports for two processes to communicate. Erasmus also offers operations for deterministic choices and nondeterministic choices by using keywords *select* and *case* respectively.

In Erasmus, communication is as important as method invocation in object-oriented languages.
The requirements of communications between two processes p_1 and p_2 are:

- p_1 must have a port, π_1, which is of protocol t_1,
- p_2 must have a port, π_2, which is of protocol t_2,
- Each protocol may contain several different types of requests, which specifies the types of requests the port can send or receive,
- There exists a channel, x, which is defined with either protocol t_1 or t_2. A channel has two ends, one is channel in for receiving incoming requests and the other is channel out for sending outgoing requests,
- The *ProcessesCommunication* property: Requests are sent by a process through its client port (declared with '–'), then received at channel in of a channel and sent out by channel out of the channel, finally received by the other process at the server port (declared with '+').
- The *Protocols* property: Given a client port π_1 of protocol t_1 and a server port π_2 of protocol t_2, if π_1 and π_2 can communicate, t_2 must satisfy t_1. Here, t_2 satisfies t_1 is defined as that the set of types of requests of t_1 must be a subset of the set of types of requests of t_2.

Some research is proposed to study communications in Erasmus, which includes constructing a fair protocol that allows arbitrary, nondeterministic communication between processes [26], describ- ing an alternative construct that allows a process to nondeterministically choose between possible communications on several channels [27], and building a static analyzer to detect communication errors between processes [28]. In this thesis, we are exploring an approach to verify consistency of communications between design and implementation of concurrent systems developed by Erasmus.

Verification Techniques

Verification techniques check whether a system conforms to its expected properties [10]. Several techniques of verification have been proposed over the years [9]. Usually, these techniques are categorized as follows [12]: *theorem proving*, *model checking*, and *static analysis*.

Theorem Proving is based on the deductive logic proposed by Floyd and Hoare [29, 30]. In this technique, a specification notation with formal semantics, along with a deductive apparatus for reasoning, are used for analysis of the program [16]. However, theorem proving requires signifi- cant mathematical calculations to analyze programs, and the process of analyzing is difficult to be automated.

Model checking is for determining if a model of a system satisfies a correctness property [9]. A model of a program consists of states and transitions, and a property is a logical formula [9]. Model checking explores all the possible states and transition of the system. If the property does not hold, the model checking algorithm generates a counterexample, an execution trace leading to a state in which the property is violated [13]. As the state space of software programs is typically too large to be analyzed completely, a major obstacle for model checking is the state space explosion problem [9, 10].

In static analysis the programs are analysed to produce useful information without executing them [31]. Static analysis has been used to detect errors which might lead to premature termina- tion or ill-defined results of the program [32]. In classical static analysis four main approaches to program analysis are introduced [33]: *data flow analysis*, *constraint based analysis*, *type and effect systems*, and *abstract interpretation*. One of the important ideas behind static analysis is abstrac- tion, which transforms a program, called concrete program, into a simpler program, called abstract program, with some key properties of the concrete program [34]. In this research, static analysis is used to extract process communications from implementation.

Galois Connection in Abstract Interpretation

Abstract interpretation is a method for gathering information about the behavior of the program from abstract semantics of the program instead of concrete semantics of the program [35]. It uses *Galois connections* to build relationships between concrete and abstract semantics with providing sound answers to questions about the behaviors of the programs [36]. Specifically, Galois connection is a relation between two partially ordered sets in order theory [35]. Given $\langle C, \pm\rangle$ and $\langle A, “\rangle$ are two partially ordered sets, and two monotone functions $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$. Then $(\alpha; \gamma)$ is a Galois connection of C and A if and only if for all $x \in C$ and $y \in A$, $\alpha(x) \pm y \equiv x “ \gamma(y)$.

Using Galois connection in abstract interpretation, the concurrent systems could be simplified as abstract models while retaining some of the properties of the systems [16]. For concurrent systems developed by Erasmus, Galois connection is exploited to build abstract semantics of systems in terms of event order vector [16, 28]. Moreover, the concept of a Galois connection is captured in category theory [37]. In our research, we make use of Galois connection to construct abstract semantics based on processes and communications of concurrent systems.

Data Flow Analysis

Given a program, it is often desirable to know the relationships between the use of values and the definition of values. Such relationships refer to the define/use relationships [38]. Data flow analysis is a static analysis technique that focuses on the information about the possible values of variables at each program point [16]. With the concept of data flow analysis, a program is allowed to be represented by data flow graphs consisting of a set of nodes and a set of edges between nodes [39]. Data flow analysis was first introduced by Kildall [40], and later was formalized by Clarke to analyze the define/use relationships [41]. In the define/use data flow technique, a program and a set of variables are analyzed according to the flow of value from the point where it is defined to the point where it is used.

For analyzing concurrent systems, a considerable amount of literature has been published on da- ta flow analysis. These include verifying the properties of systems [42], computing a set of potential static deadlock cycles for Ada tasking programs [43], using the rendezvous model of synchroniza- tion [44], studying the causal dependencies of events [45], detecting data races [46], and unifying data flow models [47]. In this research, data flow is used to analyze process communications in implementation.

Category Theory

Due to its abstractness and generality, category theory has led to its use as a conceptual frame- work in many areas of computer science [48] and software engineering [49]. It is suggested that category theory can be helpful towards discovering and verifying connections in different areas, while preserving structures in those areas [50]. In software engineering, category theory is pro- posed as an approach to formalizing refinement from design to implementation that are at different level of abstraction [14, 49]. Specifically, for modeling concurrency, category theory is used to model, analyze, and compare *Transition System*, *Trace Language*, *Event Structure*, *Petri Nets*, and other classical models of concurrency [51, 52, 53]. Besides, category theory is applied to study relationships between geometrical models for concurrency and classical models [54]. Furthermore, a categorical framework RASF has been built to formally model and verify specification, design and implementation of *Reactive Autonomic System* (RAS) [15].

However, to the best of our knowledge, there is no such kind of categorical framework for verifying the consistency between process-oriented design and implementation. The aim of this re- search is to work on the categorical framework. To understand the research, some of the categorical definitions and propositions are listed below:

Definition 2.8.1. Category: A category consists of the following components:

- Objects: A, B, C, etc.
- Morphisms: f, g, h, etc.
- Identity: For each object A there is a morphism $Id_A: A \to A$, called the identity of A.
- Domain and Codomain: For each morphism f there are given objects: $dom(f)$, $cod(f)$ called the domain and codomain of f. We write: $f: A \to B$ to indicate that $A = dom(f)$ and $B = cod(f)$.
- Composition: Given morphisms $f: A \to B$ and $g: B \to C$, i.e. with: $cod(f) = dom(g)$, there is a given morphism: $g \circ f: A \to C$, called the composite of f and g. These components are required to satisfy the following laws:
- Associativity: $h \circ (g \circ f) = (h \circ g) \circ f$, for all $f : A \to B$, $g : B \to C$, $h : C \to D$.
- Unit: $f \circ Id_A = f = Id_B \circ f$, for all $f: A \to B$.

Definition 2.8.2. Functor: A functor $F: C \to D$ between categories C and D is a mapping of objectsto objects along with morphisms to morphisms in the way of:

- $F(f: A \to B) = F(f) : F(A) \to F(B)$.
- $F(g \circ f) = F(g) \circ F(f)$; 3)
- $F(1A) = 1F(A)$.

Definition 2.8.3. Subcategory: A category C is a subcategory of a category D if:

- Every object of C is also an object of D.
- Every morphism of C is also a morphism of D.
- Composition and identities of C coincide with those of D.

Proposition 1. Poset Category: Let $(S;$ “$)$ be a poset (partially-ordered set), which satisfies reflex-
ivity, transitivity, and antisymmetry. In the poset category, each member x of S is an object; and each relation x “ y of $(S;$ “$)$ is a morphism $x \to y$.

Proof.

- Object: Each member x of S is an object of the poset category.
- Morphism: Each relation x “ y of $(S;$ “$)$ is a morphism $x \to y$.
- Identity: For every object x, there is an identity morphism x “ x, corresponding to reflexivity

in the poset.

- Composition: The morphisms (*x* “ *y*) and (*y* “ *z*) form a composition, (*y* “ *z*) ◦ (*x* “ *y*) = (*x* “ *z*), corresponding to transitivity in the poset.
- Associative: ((*x* “ *y*) ◦ (*v* “ *x*)) ◦ (*u* “ *v*) = (*v* “ *y*) ◦ (*u* “ *v*) = *u* “ *y* and (*x* “ *y*) ◦ ((*v* “ *x*) ◦ (*u* “ *v*)) = (*x* “ *y*) ◦ (*u* “ *x*) = (*u* “ *y*).

□

Summary

In this chapter, necessary background and related work for our research are introduced. Specif- ically, this chapter presents an overview of concurrent systems, explains the process-oriented lan- guages, and introduces communicating sequential processes (CSP) and Erasmus. Besides, this chap- ter give a brief introduction to verification techniques, Galois connection in abstract interpretation, data flow analysis, and some definitions in category theory.

In the next chapter, we propose an innovative categorical framework for verifying consistency of process communications between design and implementation.

Chapter 3

The Categorical Framework

Introduction

This chapter introduces the innovative categorical framework for verifying consistency of com- munications between processes. Section 3.2 briefs the contributions in developing the categorical framework. Section 3.3 illustrates the categorical framework and gives an overview of the workflow of the the framework. Section 3.4 illustrates how to design concurrent systems in CSP. Section 3.5 introduces basics of Erasmus and gives an example implemented in Erasmus. Section 3.6 describes rules for abstracting communications out of implementation, and rules for analyzing traces and failures from the abstraction. Section 3.7 explains how to construct categories based on the com- munications in the design and implementation. Section 3.8 shows approaches to construct functors between categories for verification. Section 3.9 summarizes this chapter.

Contributions

Several contributions in developing the categorical framework are introduced as follows:

- The framework for verifying process communications is proposed.
- Rules for abstracting implementation in Erasmus are proposed.
- Rules for analyzing traces and failures from abstraction of implementation in Erasmus are proposed.
- Category theory is used to model process communications in design and implementation.
- Functors are used to verify consistency of process communications between design and implementation.

The Framework

The proposed categorical framework for verification consists of the following steps (See Fig. 3.1).

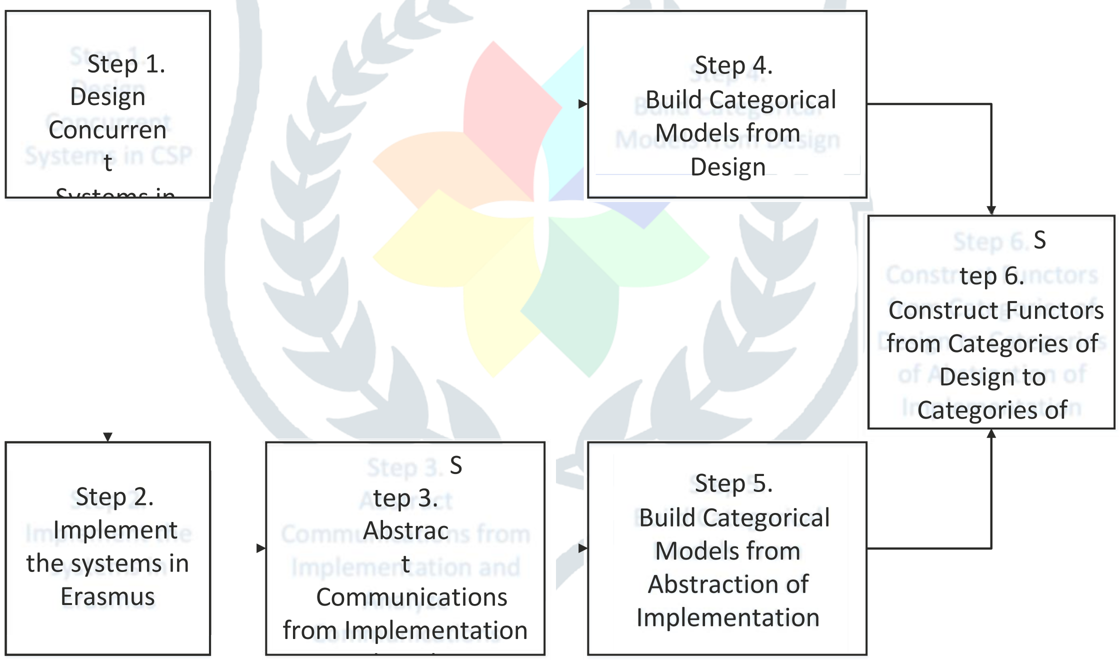

Figure 3.1: The Categorical Framework

Step 1. Design Concurrent Systems in CSP: In this step, we need to design concurrent systems in CSP, and then analyze failures of processes together with communications. This step is to achieve research objective OBJ1.

Step 2. Implement the Systems in Erasmus: In this step, we need to implement the concurrent systems in Erasmus by refining the design in step 1. This step is to achieve research objective OBJ2.

Step 3. Abstract Communications from Implementation and Analyze Communications: In this step, we need to abstract processes and communications out of the implementation in step 2, and then analyze abstract processes as well as communications. This step is to achieve research objective OBJ3.

Step 4. Build Categorical Models from Design: In this step, we need to construct categorical models for the design in step 1 with preserving structures of communications. This step is to achieve research objective OBJ4.

Step 5. Build Categorical Models from Abstraction of Implementation: In this step, we need to construct categorical models for the abstraction of implementation in step 3 with preserving structures of communications. This step is to achieve research objective OBJ5.

Step 6. Construct Functors from Categories of Design to Categories of Abstraction of Imple- mentation: In this step, we need to construct functors to verify the categorical models of the design in step 4 and the categorical model of abstraction of implementation in step 5. This step is to achieve research objective OBJ6 .

To understand the framework, the workflow of the framework is described in the following sections.

Illustration of Step 1: Design Concurrent Systems in CSP

In this research, according to CSP, a process can be represented as (alphabet, traces) and (alpha- bet, failures), where traces can represent the liveness of the process and failures can represent both liveness and safety of the process [1]. The aim of this step is to use traces and failures to design and analyze processes and communications in the concurrent system.

Traces

A trace of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time [4]. Imagine there is an observer with a notebook who watches the process and writes down the name of each event as it occurs [4]. A trace will be denoted as a sequence of symbols, separated by commas and enclosed in angle brackets

- $\langle e1, e2\rangle$ consists of two events, e1 followed by e2.
- $\langle e\rangle$ is a sequence containing only the event e.
- $\langle\rangle$ is the empty sequence containing no events.

Given two processes P and Q with alphabet A, several rules are used to derive the denotational semantics of traces of the processes [4, 5].

(1)$traces(c \rightarrow P) = \{\langle\rangle\} \cup \{\langle c\rangle^\frown t \mid t \in traces(P)\}$ (2)$traces(P ; Q) = traces(P) \cup \{s^\frown t \mid s^\frown\langle C\rangle \in P, t \in traces(Q)\}$ (3)$traces(P \text{ Q } Q) = traces(P) \cup traces(Q)$

(4) $traces(P \text{ H } Q) = traces(P) \cup traces(Q)$ (5)$traces(P \parallel Q) = traces(P) \cap traces(Q)$

In the above mentioned rules, the symbol $^\frown$ concatenate two traces, and the symbol C means the process with the trace ends successfully; (1) means that the first event in the trace is c, and followed by the events in traces of P ; (2) denotes that the traces of P ; Q come from trace of P first. When P ends successfully and Q starts to execute, the traces of P ; Q will add the traces of Q; (3) and (4) represent that the traces of P Q Q and the traces of P H Q come from traces of P or traces of Q;

(5) describes that the traces of $P \parallel Q$ come from the traces that in both traces of P and traces of Q.

Refusals and Failures

In order to distinguish between (P Q Q) and (P H Q), refusals and failures are introduced to describe processes [4, 5].

Refusals

let X be a set of events which are offered initially by the environment of a process P . If it is possible for P to deadlock on its first step when placed in this environment, we say that X is a *refusal* of P . The set of all such refusals of P is denoted by $refusals(P)$ [4].

Given two processes P and Q with alphabet A, several rules are used to derive the denotation semantics of refusals of the processes [4, 5].

(1) $refusals(c \rightarrow P) = \{X \mid X \subseteq (A - \{c\})\}$

(2) $refusals(P ; Q) = \{X \mid (X \cup \{C\}) \in refusals(P)\} \cup \{X \mid \langle C\rangle \in traces(P)\} \wedge X \in refusals(Q)\}$

(3) $refusals(P \text{ Q } Q) = refusals(P) \cap refusals(Q)$ (4)$refusals(P \text{ H } Q) = refusals(P) \cup refusals(Q)$

(5) $refusals(P \parallel Q) = \{X \cup Y \mid X \in refusals(P) \wedge Y \in refusals(Q)\}$

In the above mentioned rules, (1) means that if the first event is not c, $c \to P$ would deadlock; (2) indicates that the refusals of P ; Q are from the refusals of P first. When P ends successfully, the refusals of P ; Q are from refusals of Q; (3) describes that the refusals of P Q Q are from the refusals that would deadlock both process P and process Q; (4) represents that the refusals of P H Q are from the refusals of P or the refusals of Q, because the refusals of P and the refusals of Q can deadlock P H Q due to the nondeterminism; (5) denotes that the refusals of $P \parallel Q$ are from any set X that makes P deadlock and any set Y that makes Q deadlock.

Failures

Failures of a process is defined as a relation (set of pairs)

$$failures(P) = \{(s, X) \mid s \in traces(P) \wedge X \in refusals(P/s)\}$$

If (s, X) is a failure of P , this means that P can engage in the sequence of events recorded by s, and then refuse to do anything more, in spite of the fact that its environment is prepared to engage in any of the events of X [4, 5]. In CSP, the failures of a process usually are more informative about the behavior of that process than its traces or refusals, which can both be defined in failures as follows [4, 5].

$$traces(P) = \{s \mid \exists X \cdot (s, X) \in failures(P)\}$$

$$refusals(P) = \{X \mid (\langle\rangle, X) \in failures(P)\}$$

Given two processes P and Q with alphabet A, several rules are used to derive the denotation semantics of failures of the processes [4, 5].

(1) $failures(c \to P) = \{(\langle\rangle, X) | c \notin X\} \cup \{(\langle c\rangle^\frown s, X) \mid (s, X) \in failures(P)\}$

(2) $failures(P ; Q) = \{(s, X) | s \in A^* \wedge (s, X \cup \{C\}) \in failures(P)\}$
$\cup \{(s^\frown t, X) \mid s^\frown\langle C\rangle \in (traces)(P) \wedge (t, X) \in failures(Q)\}$

(3) $failures(P \text{ Q } Q) = \{(\langle s\rangle, X) \mid (\langle\rangle, X) \in failures(P) \cap failures(Q)$
$\vee (s \neq \langle\rangle \wedge (s, X) \in failures(P) \cup failures(Q))\}$

(4) $failures(P \text{ H } Q) = failures(P) \cup failures(Q)$

(5) $failures(P \parallel Q) = \{(s, X \cup Y) \mid s \in A^* \wedge (s, X) \in failures(P) \wedge (s, Y) \in failures(Q)\}$

In the above mentioned rules, (1) means that the failures of $c \to P$ calculate the failures when event c occurs first, and then calculate the failures of P after c; (2) indicates that the failures of P ; Q calculate the failures when process P occurs first. When P ends successfully, the failures of P ; Q depend on the failures of Q; (3) describes that when no event occurs, failures of P Q Q is the intersection of the failures of P and the failures of Q. Once the first event occurred, the failures

P Q Q depend on either the failures of P or the failures of Q; (4) represents that the failures of P H Q depends on the union of the failures of P and the failures of Q due to the nondeterminism; (5) denotes that the refusals in the failure of P and the refusals in the failure of Q together constitute the refusals in the failure of $P \parallel Q$.

Illustration of Step 2: Implement the Systems in Erasmus

In this research, Erasmus is chosen to implement concurrent systems. Erasmus is one of process- oriented programming languages. The aim of this step is to implement processes and communica- tions in Erasmus based on the design in CSP.

Erasmus

An Erasmus program consists of *cells*, *processes*, *ports*, *protocols* and *channels*. A system consists of a set of cells linked by channels. A cell, containing a collection of one or more processes or cells, provides the structuring mechanism for an Erasmus program. A process is a self-contained entity which performs computations, and communicates with other processes through its ports. A port, which is of a type of protocol, usually serves as an interface of a process for sending and receiving messages. A protocol specifies the type and the orderings of messages that can be sent and received by ports of the type of this protocol. A channel, which is of a type of protocol, must be built between two ports for two processes to communicate.

Processes and Ports

In Erasmus, processes communicate with each other through ports. Ports come in two kinds: *servers* and *clients*. Usually, a *query* is a message sent by a client to a server; a *reply* is a message sent from a server to a client. If P is a process, then $srv(P)$ is its set of server ports and $cli(P)$ is its set of client ports. Detailed definition of process and port are provided in research [22].

Messages

A message may contain data or it may be just a signal. The set of message a port can send and receive is called the *alphabet* of the port. A process may have several ports, and the alphabet of the process consists of all the messages of all its ports can send and receive. Detailed definition of message is provided in research [22].

Channel

A channel connects two ports belonging to different processes. A typical channel is a pair $\chi = (P.a, Q.b)$, where a is a port of process P and b is a port of process Q. The channel χ must have the following properties: (1). The processes P and Q must be distinct. (2). One port must be a client and the other must be a server. (3). A port must be connected to exactly one channel. Detailed definition of channel is provided in research [22].

Cells

A cell is a subsystem consisting of processes, ports, and channels. A process may be linked by channels to other processes within the cell or to ports of the cell. Cells allow us to reason about a system by separating the concerns of what happens inside a cell and what happens outside a cell. Detailed definition of cell is provided in research [22].

Protocols

A protocol determines the types and temporal sequence of values that can be communicated by a port or transmitted by a channel. Protocols are expressed as regular expressions with a few additions. For example, the protocol *Start*; (*query* ↑ *reply*)*; *Stop* means that the first message must be *Start*, then there are indefinite number of pairs of messages *query* and *reply*, and finally ends with the message *Stop*. Detailed definition of protocol is provided in research [22].

The Hello World Example

To illustrate the implementation in Erasmus, a *Hello World* example is given. The detailed syntax of Erasmus is provided in research [22]. In the following code, the message "HelloWorld" is sent from process *person* via client port *r*1 of protocol *t*1, forwarded through channel *c* of protocol *t*1, and received by process *world* via server port *r*2 of protocol *t*2. Protocol *t*1 is satisfied by protocol *t*2, as {request1: Word} is a subset of {request1: Word | request2 : Word}.

```
t1= protocol {request1:Word}
t2= protocol {request1:Word | request2:Word }


person= process r1:-t1{
        r1.request1="HelloWorld"; //sending the message to process world
}


world= process r2:+t2{
        message:Word=r2.request1; //receiving the message from process person
}
sample= cell{
          // using channel c to connect port r1 on person to port r2 on world c: Channel t1;person(c);world(c);
}
```

Illustration of Step 3: Abstract Communications from Implemen- tation and Analyze Communications

In this research, we are interested only in communications between processes. It is fundamen- tal that the code that is not related to the communications be ruled out, and the code relevant to the communications be retained. As Erasmus is based on CSP, in this research we decide to use traces and failures to analyze the semantics of Erasmus programs. The aim of this step is to use Galois connection to abstract processes and communications from the implementation, and analyze processes and communications with traces and failures in Erasmus.

Abstraction Rules

Implementation is considered as concrete domain, and abstraction of implementation is deemed as abstract domain. There are partial-order relationships, “execute before or simultaneously”, be-

tween statements in concrete domain and between statements in abstract domain respectively. There are two partial-order sets ⟨*ConcreteStatements*, ±⟩ and ⟨*AbstractStatements*, “⟩, where ± and “ represent the “execute before or simultaneously” relationship between statements in concrete do-

main and abstract domain respectively.

According to Galois Connection, relationships between statements in abstract domain must be able to be mapped to corresponding relationships between statements in concrete domain, and vice versa. Thus, there are two monotone mappings, namely α : *ConcreteStatements* → *AbstractSt- atements*, and γ : *AbstractStatements* → *ConcreteStatements*. α and γ mappings involve communication-related statements only. There are (1). for any $x, y \in$ *ConcreteStatements*, if $x \pm y$,

then $\alpha(x)$ “ $\alpha(y)$; (2). for any $a, b \in$ *AbstractStatements*, if a “ b, then $\gamma(a) \pm \gamma(b)$, and; (3). for all $x \in$ *ConcreteStatements* and $b \in$ *AbstractStatements*, $\alpha(x)$ “ $b \equiv a \pm \gamma(b)$.

The details of mapping rules for α and γ are specified in Table 3.1 and Table 3.2 respectively.

Concrete Statements	Abstract Statements
C	C
$C_1; C_2$	$C_1; C_2$
select {| a_1 | C_1 . . . | a_n | C_n }	select {a_1; C_1 | . . . | a_n; C_n}
case {‖ C_1 . . . ‖ C_n }	case {C_1 | . . . | C_n}
loop {C}	loop {C}

Table 3.1: Mapping Rules for α

Abstract Statements	Concrete Statements
C	C
$C_1; C_2$	$C_1; C_2$
select {C_1 | . . . | C_n}	select {‖ C_1 ‖ . . . ‖ C_n }
case {C_1 | . . . | C_n}	case {‖ C_1 ‖ . . . ‖ C_n }
loop {C}	loop {C}

Table 3.2: Mapping Rules for γ

In Table 3.1 and Table 3.2, C represents statements related to communications; $C_1; C_2$ means C_1 executes before C_2; | a_i | C_i ($1 \le i \le n$) in *select* indicates that if condition a_i is true, then C_i will execute (sometimes, condition a_i is not necessarily provided. If C_i is satisfied in the choice, it will be executed); ‖ is the delimiter between choices in *select* or *case* in concrete statements, while | is the delimiter between choices in *select* or *case* in abstract statements.

Analyzing Semantics of Erasmus Code

Traces and failures can be used to analyze semantics of Erasmus code.

Traces

To generate and analyze traces of processes from Erasmus implementation, several rules are defined as follows:

(1) $traces(pt.m) = \{\langle\rangle, \langle pt.m\rangle\}$

(2) $traces(pt.m_1\,;\, pt.m_2) = \{\langle\rangle, \langle pt.m_1\rangle, \langle pt.m_1, pt.m_2\rangle\}$

(3) $traces(\mathbf{loop}\{pt.m\}) = \{\langle\rangle\} \cup \{\langle pt.m\rangle^\frown t \mid t \in traces(\mathbf{loop}\{pt.m\})\}$

(4) $traces(\mathbf{case}\{pt.m_1 \mid \cdots \mid pt.m_n\}) = traces(pt.m_1) \cup \cdots \cup traces(pt.m_n)$

(5) $traces(\mathbf{select}\{pt.m_1 \mid \cdots \mid pt.m_n\}) = traces(pt.m_1) \cup \cdots \cup traces(pt.m_n)$

In the above mentioned rules, (1) means if the process sends/receives only a message m through port pt, the traces of events of this process would be empty $\langle\rangle$ and $\langle pt.m\rangle$; (2) means if the pro- cess sends/receives first message m_1 through port pt, then sends/receives the second message m_2 through port pt, the traces of events are $\{\langle\rangle, \langle pt.m_1\rangle, \langle pt.m_1, pt.m_2\rangle\}$; (3) means if the process consists of an indefinite loop of sending/receiving a message m through port pt, the traces of events would contain traces of indefinite recursion of $pt.m$; and (4) and (5) represent that deterministic and nondeterministic choices, respectively, can be modeled using the same approach as a selection among traces of events.

Failures

To generate and analyze failures of processes from Erasmus implementation, several rules are defined as follows:

(1) $failures(p.m) = \{(\langle\rangle, X) \mid X \subseteq (alphabet(p) - m)\}$

(2) $failures(C_1; C_2) = \{(s, X) \mid (s, X) \in failures(C_1)\}$

$\cup \{(s^\frown t, X) \mid s^\frown\langle C\rangle \in traces(C_1) \wedge (t, X) \in failures(C_2)\}$

(3) $failures(\mathbf{loop}\{C\}) = \{(s, X) \mid (s, X) \in failures(C)\}$

$\cup \{(s^{1\frown} s, X) \mid s^{1\frown}\langle C\rangle \in traces(C) \wedge (s, X) \in failures(C)\}$

$\cup \ldots \cup \{(s^{1\frown} s^{2\frown} \ldots {}^\frown s^{n-1\frown} s^n, X) \mid s^{i\frown}\langle C\rangle \in traces(C)$

$\wedge\ 1 \le i \le n-1 \wedge (s, X) \in (failures(C))\}$ (4) $failures(\mathbf{case}\{C_1 | \ldots$

$|C_n\}) = \{(s, X)|(s, X) \in failures(C_1) \cup \ldots \cup failures(C_n)\}$

(5) $failures(\textbf{select}\{C_1|\ldots|C_n\}) = \{(s, X)|(s = \langle\rangle \wedge (s, X) \in failures(C_1) \cap \ldots \cap failures(C_n))$

$\vee (s \;/\!= \langle\rangle \wedge (s, X) \in failures(C_1) \cup \ldots \cup failures(C_n))\}$

(6) $failures(C_1 \parallel C_2) = \{(s, X \cup Y)|((s, X) \in failures(C_1) \wedge (s, Y) \in failures(C_2))\}$

In (1), the message can be represented by *p.m*. *p.m* is a simple statement. *failures*(*p.m*) means any event occurs on port *p* other than message *m*, *p* stops working. In (2), let C_1 and C_2 be two state- ments , and let C_1 execute before C_2. $failures(C_1; C_2)$ means that the failures become $failures(C_1)$ first. After C_1 accomplishing its execution successfully, the failures depend on $failures(C_2)$. In (3), let C be a statement iterating n times in a loop, and let C^i represent the *ith* iteration of a loop of

C. $failures(\textbf{loop}\{C\})$ means that if C iterates once, the failures become $failures(C)$; if C iterates twice, and if the execution of the first iteration is accomplished successfully with trace s^1, the fail- ures depend on $failures(C)$ in the second iteration; if C iterates n times, and if the execution from 1*st* iteration to $(n-1)$*th* iteration successfully with trace $s^{1\frown}s^{2\frown}\ldots{}^\frown s^{n-1}$, the failures depend on

$failures(C)$ in the *nth* iteration. In (4), let C_i be a statement where $1 \leq i \leq n$, and let *case* represent nondeterministic choices. $failures(\textbf{case}\{C_1 | \ldots | C_n\})$ means that the failures depend on one of $failures(C_i)$ where $1 \leq i \leq n$. In (5), let C_i be a statement where $1 \leq i \leq n$, and let *select* repre- sent deterministic choices. $failures(\textbf{select}\{C_1 | \ldots | C_n\})$ means that if statements $C_1 \ldots C_n$ wait for the occurrence of the first message, the failures become $failures(C_1) \cap \ldots \cap failures(C_n)$. Whenthe trace s occurs, it indicates C_i executes, so the failures are in $failures(C_1) \cup \ldots \cup failures(C_n)$. In (6), let C_1 be a statement from a process, let C_2 be a statement from another process, and let C_1 and C_2 be able to communicate with each other. In Erasmus, two ports can communicate only when the same message is sent by a port and received by another port simultaneously. If there is a failure of $C_1 \parallel C_2$, the failure would be from $failures(C_1)$ and $failures(C_2)$.

Illustration of Step 4 and Step 5: Build Categorical Models from Design and Abstraction of Implementation

In this research, category theory is used to model communications and processes in the design and the abstraction of implementation. The aim of these two steps is to construct categories for modeling communications in the design and the abstraction of implementation.

To construct categorical models of traces and failures of processes and communications, several definitions are provided as follows.

Proposition 2. Category of Traces: Each object is a set of traces to indicate a process. A mor- phism $traces(A) \to traces(B)$ means traces of process A evolves to traces of process B, where $traces(A) \subseteq traces(B)$.

Proof.

Objects: Each object is a set of traces of events. Such as $\{\langle\rangle, \langle sq\rangle, \langle sq, tq\rangle, \ldots\}$.

Morphisms: Let $traces(A)$ and $traces(B)$ be objects. If $traces(A) \subseteq traces(B)$, there is a morphism $traces(A) \to traces(B)$.

Identities: For each object, $traces(A)$, there is an identity $traces(A) \to traces(A)$, which indicates $trace(A) \subseteq traces(B)$.

Composition: Given any morphisms $morph_{A,B} : traces(A) \to traces(B)$ and $morph_{B,C} : traces(B) \to traces(C)$, with codomain of $morph_{A,B}$ = domain of $morph_{B,C}$, there is $traces(A) \subseteq traces(B) \subseteq traces(C)$. Thus, there is a composition morphism: $morph_{B,C} \circ morph_{A,B} : traces(A) \to traces(C)$, which means $traces(A) \subseteq traces(C)$.

Associativity: For all morphisms $morph_{A,B} : traces(A) \to traces(B)$, $morph_{B,C} : traces(B) \to traces(C)$ and $morph_{C,D} : traces(C) \to traces(D)$, with codomain of $morph_{A,B}$ = domain of $morph_{B,C}$ and codomain $morph_{B,C}$ = domain of $morph_{C,D}$, there is $traces(A) \subseteq traces(B) \subseteq traces(C) \subseteq trace(D)$. Thus, there are $morph_{C,D} \circ (morph_{B,C} \circ morph_{A,B}) = morph_{C,D} \circ (traces(A) \to trace(C)) = traces(A) \to traces(D)$, and $(morph_{C,D} \circ morph_{B,C}) \circ morph_{A,B} = (traces(B) \to traces(D)) \circ morph_{A,B} = traces(A) \to traces(D)$. So, $morph_{C,D} \circ (morph_{B,C} \circ morph_{A,B}) = (morph_{C,D} \circ morph_{B,C}) \circ morph_{A,B}$

Proposition 3. Category of Failures: Each object is of the form *failures* to indicate a process. A Morphism $failures_a \to failures_b$ means the process with the failures from trace $\langle\rangle$ to the trace a evolves to the process with the failures from trace $\langle\rangle$ to the trace b, where $failures_a \subseteq failures_b$.

Proof.

Objects: Each object is failures of a process. For example, $failures_{\langle e1 \ldots e2\rangle}$ represents all the failures from trace $\langle\rangle$ to trace $\langle e1 \ldots e2\rangle$. $failures_{\langle\rangle} = \{(\langle\rangle, X) \mid \langle\rangle \in traces(P) \wedge X \in refusals(P/\langle\rangle)\}$ is an object, $failures_{\langle e1\rangle} = \{\{(\langle\rangle, X) \mid \langle\rangle \in traces(P) \wedge X \in refusals(P/\langle\rangle)\}$, $\{(\langle e1\rangle, X) \mid \langle e1\rangle \in traces(P) \wedge X \in refusals(P/\langle e1\rangle)\}\}$ is an object as well.

Morphisms: Let $failures_x$ and $failure_y$ be objects. If $failures_x \subseteq failures_y$, there is a morphism $failures_x \to failures_y$. It means process of $failures_x$ evolves to $failures_y$. For example, there is a morphism $failures_{\langle\rangle} \to failures_{\langle e1\rangle}$.

Identities: For each object, $failures_m$, there is an identity $failures_m \to failures_m$, which indi- cates $failures_m \subseteq failures_m$. For example, there is a morphism $failures_{e\langle 1\rangle} \to failures_{e\langle 1\rangle}$.

Composition: Given any morphisms $morph_{x,y} : failures_x \to failures_y$ and $morph_{y,z} : failures_y \to failures_z$, with codomain of $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_x \subseteq failures_y \subseteq failures_z$. Thus, there is a composition morphism: $morph_{y,z} \circ morph_{x,y} : failures_x \to failures_z$.

Associativity: For all morphisms $morph_{w,x} : failures_w \to failures_x$, $morph_{x,y} : failures_x \to failures_y$ and $morph_{y,z} : failures_y \to failures_z$, with codomain of $morph_{w,x}$ = domain of $morph_{x,y}$ and codomain

$morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_w \subseteq failures_x \subseteq failures_y \subseteq failures_z$ to represent the subset relationships between failures. Thus, there are $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}) = morph_{y,z} \circ (failures_w \rightarrow failures_y) = failures_w \rightarrow failures_z$, and $(morph_{y,z} \circ morph_{x,y}) \circ morph_{w,x} = (failures_x \rightarrow failures_z) \circ morph_{w,x} = failures_w \rightarrow failures_z$. So, $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}) = (morph_{y,z} \circ morph_{x,y}) \circ morph_{w,x}$.

□

Illustration of Step 6: Construct Functors from Categories of De- sign to Categories of Abstraction of Implementation

In this research, we focus on the consistency of process communications between design and implementation. The aim of this step is to use category theory to verify the consistency of process communications between design and implementation.

To understand the consistency of process communications between design and implementation, several definitions are provided as follows.

Definition 3.8.1. Consistency of Communications with Traces: Given a sequence of sets of traces in the design representing the progress of the system, $DTraces : \{\langle\rangle\} \rightarrow \{\langle\rangle, \langle devent_1 \rangle\} \rightarrow \cdots \rightarrow \{\langle\rangle, \langle devent_1 \rangle, \cdots, \langle devent_1, \ldots, devent_n \rangle\}$, and a sequence of traces in the implemen- tation representing the progress of the system, $ITraces : \{\langle\rangle\} \rightarrow \{\langle\rangle, \langle ievent_1 \rangle\} \rightarrow \cdots \rightarrow \{\langle\rangle, \langle ievent_1 \rangle, \cdots, \langle ievent_1, \ldots, ievent_n \rangle\}$. If there exists a mapping from *DTraces* to *ITtraces* with sequence preserved, which can map $\{\langle\rangle, \langle devent_1 \rangle, \cdots, \langle devent_1, \ldots, devent_i \rangle\}$ to $\{\langle\rangle, \langle ievent_1 \rangle, \cdots, \langle ievent_1, \ldots, ievent_i \rangle\}$, and $\{\langle\rangle, \langle devent_1 \rangle, \cdots, \langle devent_1, \ldots, devent_i, devent_{i+1} \rangle\}$ to $\{\langle\rangle, \langle ievent_1 \rangle, \cdots, \langle ievent_1, \ldots, ievent_i, ievent_{i+1} \rangle\}$, then, *ITraces* is consistent with *DTraces*. If all sequences in the design have corresponding mapping sequences in the implementation, the commu- nications in the implementation are consistent with the communications in the design.

Definition 3.8.2. Consistency of Communications with Failures: Given a sequence of commu- nications with failures in the design to represent the progress of communications, $DFailures : failures_{\langle\rangle} \rightarrow failures_{\langle devent_1 \rangle} \rightarrow \cdots \rightarrow failures_{\langle devent_1, \ldots, devent_n \rangle}$, and a sequence of communica- tions with failures in the implementation to represent the progress of communications, $IFailures : failures_{\langle\rangle} \rightarrow failures_{\langle ievent_1 \rangle} \rightarrow \cdots \rightarrow failures_{\langle ievent_1, \ldots, ievent_n \rangle}$. If there exists a mapping from *DFailures* to *IFailures* with structure preserved between failures, which can map each trace of

$failures_{\langle devent1, ..., devent_i \rangle}$ to the same trace of $failures_{\langle ievent1, ..., ievent_i \rangle}$ with the refusals of the trace of $failures_{\langle devent1, ..., devent_i \rangle}$ being a subset of the refusals of the corresponding trace of $failures_{\langle ievent_1, ..., ievent_i \rangle}$, and can map $failures_{\langle devent_1, ..., devent_i \rangle} \rightarrow failures_{\langle devent_1, ..., devent_{i+1} \rangle}$ to $failures_{\langle ievent1, ..., ievent_i \rangle} \rightarrow failures_{\langle ievent1, ..., ievent_{i+1} \rangle}$, then *IFailures* is consistent with *DFailures*. If all sequences in the design have corresponding mapping sequences in the implementation, the communications in the implementation are consistent with the communications in the design.

As functor can be used to check structure preserving between two categories, in this research, functors are used to verify consistency of communications with traces and failures between design and implementation [55, 56, 57, 58]. Successful construction of such functor means the process communications in the implementation is consistent with the process communications in the de- sign. Failing to construct such functor could indicate an inconsistency between the design and the implementation.

To construct functors from categories of traces in design to categories of traces in abstraction of implementation, an approach for the construction is introduced as follows.

- For each object, *ocd*, in design, there must be a corresponding object, *oci*, in implementation, such that *ocd* can be mapped to *oci* when each trace in *ocd* has the same trace in *oci*.
- For each morphism $md : ocd1 \rightarrow ocd2$ in design, there must be a corresponding morphism $mi : oci1 \rightarrow oci2$ in implementation, such that *md* can be mapped to *mi* when *ocd1* and *ocd2* can be mapped to *oci1* and *oci2* respectively.

To construct functors from categories of failures in design to categories of failures in implementation, an approach for the construction is introduced as follows.

- For each object, *ocd*, in design, there must be a corresponding object, *oci*, in implementation, such that *ocd* can be mapped to *oci* when each trace in *ocd* has the same trace in *oci*, and the corresponding refusals in *ocd* are a subset of the corresponding refusals in *oci*.
- For each morphism $md : ocd1 \rightarrow ocd2$ in design, there must be a corresponding morphism $mi : oci1 \rightarrow oci2$ in implementation, such that *md* can be mapped to *mi* when *ocd1* and *ocd2* can be mapped to *oci1* and *oci2* respectively.

Summary

In this chapter, we propose the innovative categorical framework to verify consistency of process communications between design and implementation. The workflow of the framework consists of 6 steps. In the step 1, we use traces and failures in CSP to model and analyze design of concurrent systems. In step 2, we use Erasmus to implement concurrent systems. In step 3, we use Galois connections to abstract process communications out of implementation, and define rules to analyze traces and failures from the abstraction of implementation. In step 4 and step 5, we use categories of traces and categories of failures to model design and abstraction of implementation. Finally, in step 6, we propose approaches to construct functors between categories for verification.

In the next chapter, we introduce how to use the categorical framework to verify consistency of communications traces between design and implementation.

Chapter 4

Verifying Communications with Traces

Introduction

A process can be modeled in terms of traces that can represent the liveness of the process. In this chapter, by using the categorical framework, we can verify consistency of communications with traces between design and implementation. Section 4.2 briefs the contributions in verifying communications with traces. Section 4.3 introduces the categorical framework for verifying com- munications with traces between design and implementation. Section 4.4 gives an overview of a running example to illustrate the application of the framework for verification with traces. Sec- tion 4.5 summarizes this chapter.

Contributions

Several contributions in verifying communications with traces are introduced as follows:

- The framework for verification with traces is proposed.
- Category theory is used to model communications with traces in design and implementation.
- Functors are used to verify consistency of communications with traces between design and implementation.

The Framework for Verification with Traces

As stated in Chapter 3, we apply the framework described in Chapter 3 to model and analyze the consistency of communications with traces. Fig. 5.1 depicts the process of communication ver- ification with traces in the categorical framework. The steps of the verification process are outlined next.

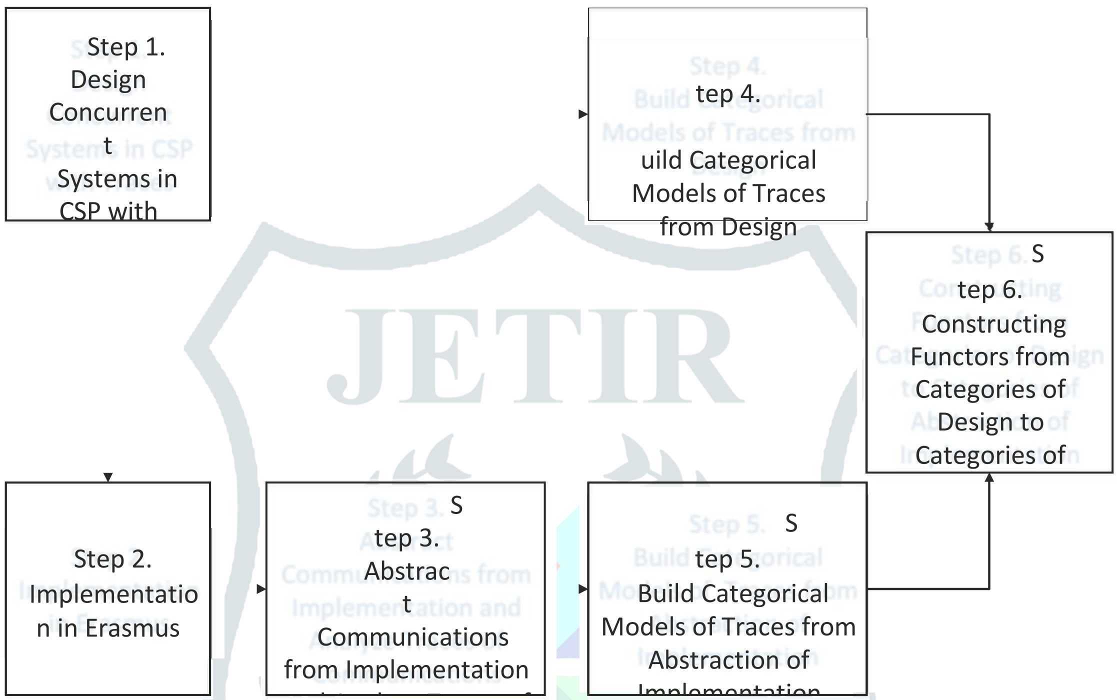

Figure 4.1: The Categorical Framework for Verification with Traces

Step 1. Design Concurrent Systems in CSP with Traces: In this step, we need to design concur- rent systems in CSP, and then analyze traces of processes together with communications. This step is to achieve research objective OBJ1.

Step 2. Implement the Systems in Erasmus: In this step, we need to implement the concurrent systems in Erasmus by refining the design in step 1. This step is to achieve research objective OBJ2. Step 3. Abstract Communications from Implementation and Analyze Traces of Communica- tions: In this step, we need to abstract processes and communications out of implementation in step 2, and then, analyze traces of abstract processes as well as communications. This step is to achieve research objective OBJ3.

Step 4. Build Categorical Models of Traces from Design: In this step, we need to construct categorical models for the design in step 1 with preserving structures of communications. This step is to achieve research objective OBJ4.

Step 5. Build Categorical Models of Traces from Abstraction of Implementation: In this step, we need to construct categorical models for the abstraction of implementation in step 3 with preserving structures of communications. This step is to achieve research objective OBJ5.

Step 6. Construct Functors from Categories of Design to Categories of Abstraction of Imple- mentation: In this step, we need to construct functors to verify the categorical models of the design in step 4 and the categorical model of abstraction of implementation in step 5. This step is to achieve research objective OBJ6.

To illustrate the application of the framework for verification with traces, the workflow of the framework are described by a running example in the following sections.

Illustration of a Running Example

To illustrate the framework, an example with three processes *Student*, *TeachingAssistant* and *Professor* is developed. These processes collaborate as a concurrent system to deal with questions and answers as the following steps:

(1) *Student* asks *TeachingAssistant* a question.

(2) If *TeachingAssistant* can answer the question, the answer will be given to *Student*. Otherwise, *TeachingAssistant* will forward the question to *Professor*.

(3) Once *Professor* receives the question, it will give the answer to *TeachingAssistant*, and then *TeachingAssistant* will forward the answer to *Student*.

(4) steps 1,2,3 can repeat indefinitely.

In the requirements, there are two communication scenarios. In the first scenario, the *TeachingAssis- tant* can answer the question. In the second scenario, *Professor* helps *TeachingAssistant* to answer the question.

Illustration of Step 1: Design Concurrent Systems in CSP with Traces

The aim of this step is to design and analyze the processes and the concurrent system in CSP based on the textual description of the system requirements.

Step1.a: Model the Conceptual Design

As CSP can model and specify processes in concurrent system, for this example, the design of the above described system is specified as follows:

$$Stud = sq \to ta \to Stud \quad Prof = tq \to pa \to Prof$$

$$TA = ((sq \to ta \to TA) \text{ H } (sq \to tq \to TA)) \text{ Q } (pa \to ta \to TA)$$

In this design, event *sq* indicates the question asked by *Student* to *TeachingAssistant*; event *ta* repre- sents the answer given by *TeachingAssistant* to *Student*; event *tq* stands for the question forwarded by *TeachingAssistant* to *Professor*; event *pa* describes the answer given by *Professor* to *Teachin- gAssistant*; → denotes the “occurs before” relation between events; H means the nondeterministic choices made by the process itself; and Q stands for the deterministic choices based on the event from the environment.

Step1.b: Generate and Analyze Traces

Traces in CSP are used to analyze behaviors of a concurrent system. A trace of events represents a sequential record of the behavior of a process. A process behaves in different ways leading to different traces of events.

To generate and analyze traces of processes in CSP, according to Chapter 3, the following rules defined in CSP [4, 5] are used in this research.

$$traces(c \to P) = \{\langle\rangle\} \cup \{\langle c\rangle^\frown t \mid t \in traces(P)\}$$

$$traces(P ; Q) = traces(P) \cup \{s^\frown t \mid s^\frown\langle \text{C}\rangle \in P, t \in traces(Q)\}$$

$$traces(P \text{ Q } Q) = traces(P) \cup traces(Q) \quad traces(P \text{ H } Q) = traces(P) \cup traces(Q)$$

$$traces(P \parallel Q) = traces(P) \cap traces(Q)$$

Model Individual Processes with Traces

For the above mentioned example, all possible traces of each process *Student*, *TeachingAssis- tant*, and *Professor* can be generated, analyzed and represented from the CSP specification of the design as follows:

$$traces(Stud) = \{\langle\rangle, \langle sq\rangle, \langle sq, ta\rangle^\frown t \mid t \in traces(Stud)\} \quad traces(Prof) = \{\langle\rangle, \langle tq\rangle, \langle tq, pa\rangle^\frown t \mid t \in traces(Prof)\} \quad traces(TA) = \{\langle\rangle, \langle sq\rangle, \langle sq, ta\rangle^\frown t \mid t \in$$

traces(*TA*)}

$$\cup \ \{\langle\rangle, \langle sq \rangle, \langle sq, tq\rangle^\frown t \mid t \in traces(TA)\}$$

$$\cup \ \{\{\langle\rangle, \langle pa\rangle, \langle pa, ta\rangle^\frown t \mid t \in traces(TA)\}$$

In this listing of traces, the function *traces* stands for generating a set of all possible traces; t in $t \in traces(P)$ is one of the traces of process P ; $\langle event_1, \cdots, event_n\rangle$ indicates the a specific trace of events; $^\frown$ concatenates two traces into one; and $\{traces_1\} \cup$ behave as either $\{traces_1\}$ or $\{traces_2\}$.

Model Communications between Processes with Traces

$\{traces_2\}$ denotes the process may

When processes *Student*, *TeachingAssistant*, and *Professor* work in parallel as a system, CSP operator "‖" models communication between processes. According to CSP, if there is a communi- cation between two processes, there must be an event that occurs in both processes simultaneously. The set of all possible traces of the system can be generated, analyzed and represented from the

CSP specification of the design as follows:

$$traces(Stud \parallel TA \parallel Prof\,) =$$

$$\{\langle\rangle, \langle sq \rangle, \langle sq, ta\rangle^\frown t \mid t \in traces\ (Stud \parallel TA \parallel Prof\,)\}$$

$$\cup \ \{\langle\rangle, \langle sq \rangle, \langle sq, tq \rangle, \langle sq, tq, pa\rangle, \langle sq, tq, pa, ta\rangle^\frown t \mid t \in traces(Stud \parallel TA \parallel Prof\,)\}$$

According to the generated traces of events of processes running in parallel, the system should behave as either *TeachingAssistant* answers the question from *Student* directly, or *TeachingAssistant* asks help from *Professor* to answer *Student*.

Fig.4.2 shows a representation of $traces(Stud \parallel TA \parallel Prof\,)$ as a directed graph:

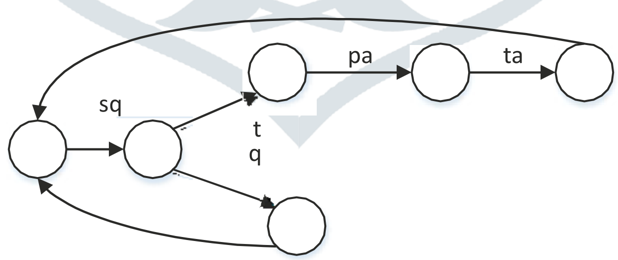

Figure 4.2: Traces of Communications between Student, TeachingAssistant and Professor

Illustration of Step 2: Implement the systems in Erasmus

The aim of this step is to implement the processes and the concurrent system in Erasmus based on the design.

In this implementation, there are two scenarios: *TeachingAssistant* answering the question from *Student*, and *TeachingAssistant* resorting to help from *Professor* to answer the question from *Stu- dent*. To communicate with each other, two processes need to build a channel between their ports. For example, process *Student* can ask a *question* through port *s*, then the *question* passes through the channel *SQuestion*, and the *question* is received on port *s* by process *TeachingAssistant*.

The Erasmus implementation is as follows.

```
Prot = protocol { squestion | tanswer | tquestion | panswser }
//accept question or answer

Student= process -s:Prot, +t:Prot { loop {
    s.squestion;//ask the question via port s t.tanswer; //receive the answer via port t
  }
}

TeachingAssistant = process +s:Prot, -t:Prot, +p:Prot, -t':Prot { loop
  select{          //deterministic choices depend on the environment
    ||s.squestion; //receive the question from Student via port s case{   //nondeterministic   choices   made   by   the
        process
          || t.tanswer;                     //send the answer to Student via port t
          || t'.tquestion; }                      //ask the question to Professor via port t'
    ||p.panswer; //receive the answer from Professor via port p t.tanswer; //send the answer to Student via port t
  }
}
Professor = process +t':Prot, -p:Prot { loop{
    t'.tquestion; //receive the question from TeachingAssistant p.panswser;      //send      the      answer      to
    TeachingAssistant
  }
```

```
}

System = cell{                    //encapsulate processes
    // channels to connect ports
    SQuestion, TAnswer, T'Question, PAnswer: Prot; Student(SQuestion,TAnswer);
    TeachingAssistant(SQuestion,TAnswer,PAnswer,T'Question); Professor(T'Question,PAnswer);
}
```

Illustration of Step 3: Abstract Communications from Implementation and Analyze Traces of Communications

Since the interest in this thesis is in analyzing the behaviors of the system based on traces of events, an abstraction is created for extracting the code pertaining to generate traces of events. The aim of this step is to use Galois connection to abstract processes and communications from the implementation, and analyze processes and communications with traces in Erasmus.

Step3.a: Abstract the Implementation

According to the abstraction rules in Chapter 3, the abstraction of implementation contains loops, deterministic choices, nondeterministic choices, sending and receiving messages through ports. The abstraction of the Erasmus implementation is provided as follows.

```
Student =
loop{
            s.squestion;
            t.tanswer
      }

TeachingAssistant=
      loop
```

```
        select { (
            s.squestion;
            case{
                t.tanswer
                |t'.tquestion
            }
        )
|{ p.panswer ;
t.tanswer}
    }
 Professor=
    loop{
        t'.tquestion;
        p.panswer
    }
```

In the above mentioned abstraction of implementation, **loop** can be defined by recursion; **selec- t** together with | represent deterministic choices; **case** together with | represent nondeterminis- tic choices; the notation *PROCESS.port.message*(for example *TeachingAssistant.s.squestion*) rep- resents *message*(*squestion*) that occurs on *PROCESS*(*TeachingAssistant*) through *port*(*s*); and the symbol ";" is the operator to indicate the "occurs before" relation between messages.

In this example, implementation is considered as concrete domain, and abstraction is considered as abstract domain. The relationships "execute before or simultaneously" between statements in abstraction are maintained in implementation, and vice versa. The details of mappings for the example are shown in Fig. 4.4.

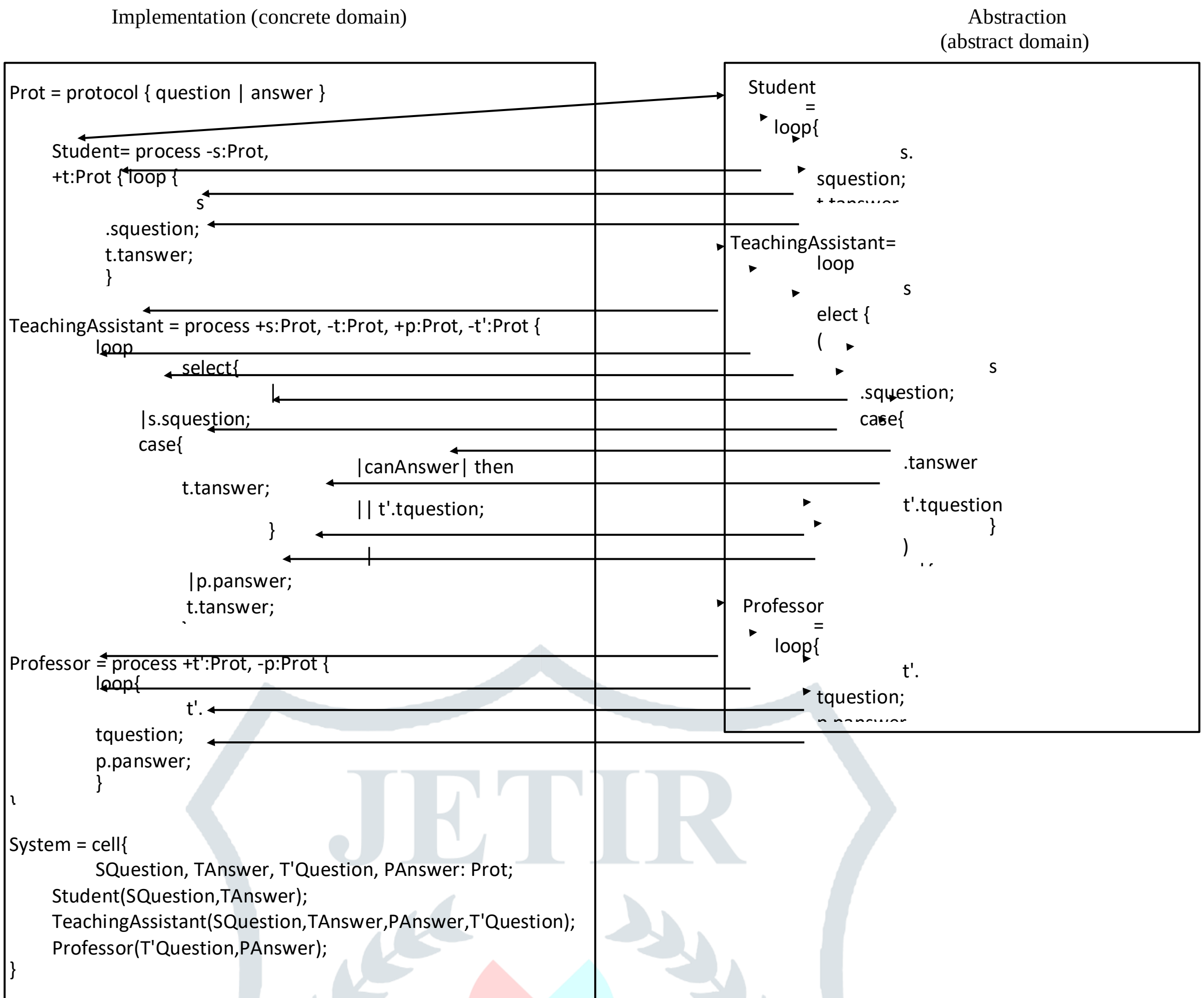

Figure 4.4: Mappings Between Implementation and Abstraction of the Student, Teaching Assistant and Professor Example

Step3.b: Generate and Analyze Traces

Although the syntax of Erasmus is different from CSP, the semantics of Erasmus is analogous to CSP. Some notations that model traces of events in CSP can be also used to model traces of events in Erasmus with preserving the same syntax and semantics, which includes $\frown$, $\cup$, $\langle\rangle$, H and Q. Like CSP, *traces* in Erasmus does not distinguish H from Q.

To generate and analyze the traces of processes in Erasmus, according to Chapter 3, the following rules are used in this research.

(1)$traces(pt.m) = \{\langle\rangle, \langle pt.m\rangle\}$

(2)$traces(pt.m_1 ; pt.m_2) = \{\langle\rangle, \langle pt.m_1 \rangle, \langle pt.m_1 , pt.m_2 \rangle\}$

(3)$traces(\mathbf{loop}\{pt.m\}) = \{\langle\rangle\} \cup \{\langle pt .m\rangle^\frown t \mid t \in traces(\mathbf{loop}\{pt.m\})\}$ (4)$traces(\mathbf{case}\{pt.m_1$

$\mid \cdots \mid pt.m_n \}) = traces(pt.m_1) \cup \cdots \cup traces(pt.m_n)$ (5)$traces(\mathbf{select}\{pt.m_1 \mid \cdots \mid pt.m_n \}) =$

$traces(pt.m_1) \cup \cdots \cup traces(pt.m_n)$

For each process in the abstract implementation, the traces of events are generated and analyzed as follows.

$traces(Student) =traces(\mathbf{loop}\ \{s.squestion; t .tanswer \})$

$= \{\langle\rangle, \langle s .squestion\rangle, \langle s .squestion, t .tanswer \rangle^\frown t \mid t \in traces(Student)\},$

$traces(TeachingAssistant) =$

$traces(\mathbf{loop\ select}\ \{(s.squestion; \mathbf{case}\ \{t.tanswer \mid t^J.tquestion\}) \mid (p.panswer ; t .tanswer)\})$

$= \{\{\langle\rangle, \langle s .squestion\rangle, \langle s .squestion, t .tanswer \rangle^\frown t \mid t \in traces(TeachingAssistant)\}\}$

$\cup \{\{\langle\rangle, \langle s .squestion\rangle, \langle s .squestion, t^J.tquestion\rangle^\frown t \mid t \in traces(TeachingAssistant)\}\}$

$\cup \{\{\langle\rangle, \langle p.panswer \rangle, \langle p.panswer, t .tanswer \rangle^\frown t \mid t \in traces\ (TeachingAssistant)\}\},$

$traces(Professor) =traces(\mathbf{loop}\ \{t^J.tquestion; p.panswer \})$

$= \{\langle\rangle, \langle t^J.tquestion\rangle, \langle t^J.tquestion, p.panswer \rangle^\frown t \mid t \in traces(Professor)\}.$

In the implementation, when one process communicates a message with another process, there is an event that occurs simultaneously on both processes. In the above implementation of the example, ports with the same name in different processes are connected by a channel. For example, port *s* of process *Student* can send a question to port *s* of process *TeachingAssistant*, and there is an event *squestion*. The event *squestion* occurs on both *Student* and *TeachingAssistant* simultaneously during the communication. The two ports are connected by channel *SQuestion* according to the implementation.

To generate and analyze the traces of concurrent systems in the implementation in Erasmus, the function *traces*() together with the symbol ∥ are defined as follows.

(1) Given a process P with port pt_1 and a process Q with port pt_2, pt_1 and pt_2 are connected to a channel ch. P has a trace p, and Q has a trace q. The head of p and q are events p_0 and q_0 respectively, and the tail of p and q are traces p' and q' respectively. When p and q run in parallel,

$$traces(P \parallel Q) = \{\langle\rangle\} \cup \{\langle m_1\rangle^\frown t \mid m_1 = p_0, m_1 = q_0, t \in traces(P/p_0 \parallel Q/q_0)\}$$

(2) Given processes $P_1, \cdots, P_n$, when they run in parallel as a system,

$$traces(P_1 \parallel \cdots \parallel P_n) = traces(P_1) \parallel \cdots \parallel traces(P_n)$$

For each of the two scenarios for the system in the implementation, the traces of events are generated and analyzed as follows.

Scenario 1: *TeachingAssistant* answers the question.

$$
\begin{aligned}
&traces(Student \parallel TeachingAssistant \parallel Professor)\\
&= \{\langle\rangle, \langle squestion\rangle, \langle squestion, tanswer\rangle^\frown t \mid t \in traces(Student)\}\\
&\parallel \{\langle\rangle, \langle squestion\rangle, \langle squestion, tanswer\rangle^\frown t \mid t \in traces(TeachingAssistant)\}\\
&\cup \{\langle\rangle, \langle squestion\rangle, \langle squestion, tquestion\rangle^\frown t \mid t \in traces(TeachingAssistant)\}\\
&\cup \{\langle\rangle, \langle panswer\rangle, \langle panswer, tanswer\rangle^\frown t \mid t \in traces(TeachingAssistant)\} \parallel \{\langle\rangle,\\
&\langle tquestion\rangle, \langle tquestion, panswer\rangle^\frown t \mid t \in traces(Professor)\}\\
&= \{\langle\rangle, \langle squestion\rangle^\frown s \mid\\
&s \in \{\{\langle tanswer\rangle^\frown t \mid t \in traces(Student)\}\\
&\parallel \{\langle tanswer\rangle^\frown t \mid t \in traces(TeachingAssistant)\}\\
&\cup \{\langle tquestion\rangle^\frown t \mid t \in traces(TeachingAssistant)\}\\
&\parallel \{\langle\rangle, \langle tquestion\rangle, \langle tquestion, panswer\rangle^\frown t \mid t \in traces(Professor)\}\}\}\\
&= \{\langle\rangle, \langle squestion\rangle, \langle squestion, tanswer\rangle^\frown s \mid\\
&s \in traces(Student) \parallel traces(TeachingAssistant) \parallel traces(Professor)\}
\end{aligned}
$$

In scenario 1, *Student* sends *squestion* to *TeachingAssistant*, and then *TeachingAssistant* sends *tan- swer* to *Student*.

Scenario 2: *Professor* helps *TeachingAssistant* to answer the question.

$traces(Student \parallel TeachingA \parallel Professor)$

$=\{\langle\rangle, \langle squestion\rangle, \langle squestion, tanswer\rangle^\frown t \mid t \in traces(Student)\}$

$\parallel\{\langle\rangle, \langle squestion\rangle, \langle squestion, tanswer\rangle^\frown t \mid t \in traces(TeachingAssistant)\}$

$\cup \{\langle\rangle, \langle squestion\rangle, \langle squestion, tquestion\rangle^\frown t \mid t \in traces(TeachingAssistant)\}$

$\cup \{\langle\rangle, \langle panswer\rangle, \langle panswer, tanswer\rangle^\frown t \mid t \in traces(TeachingAssistant)\} \parallel\{\langle\rangle,$

$\langle tquestion\rangle, \langle tquestion, panswer\rangle^\frown t \mid t \in traces(Professor)\}$

$=\{\langle\rangle \langle squestion\rangle^\frown s \mid$

$s \in \{\{\langle tanswer\rangle^\frown t \mid t \in traces(Student)\}$

$\parallel\{\langle tanswer\rangle^\frown t \mid t \in traces(TeachingAssistant)\}$

$\cup \{\langle tquestion\rangle^\frown t \mid t \in traces(TeachingAssistant)\}$

$\parallel\{\langle\rangle, \langle tquestion\rangle, \langle tquestion, panswer\rangle^\frown t \mid t \in traces(Professor)\}$

$=\{\langle\rangle, \langle squestion\rangle, \langle squestion, tquestion\rangle^\frown s \mid s \in$

$\{\{\langle tanswer\rangle^\frown t \mid t \in traces(Student)\}$

$\parallel traces(TeachingAssistant)$

$\parallel\{\langle panswer\rangle^\frown t \mid t \in traces(Professor)\}\}\}$

$=\{\langle\rangle, \langle squestion\rangle, \langle squestion, tquestion\rangle, \langle squestion, tquestion, panswer\rangle^\frown s \mid$

$s \in \{\{\langle tanswer\rangle^\frown t \mid t \in traces(Student)\} \parallel\{\langle tanswer\rangle^\frown t \mid t \in$

$traces(TeachingAssistant)\} \parallel traces(Professor)\}\}\}$

$=\{\langle\rangle, \langle squestion\rangle, \langle squestion, tquestion\rangle, \langle squestion, tquestion, panswer\rangle,$

$\langle squestion, tquestion, panswer, tanswer\rangle^\frown s \mid$

$s \in traces(Student) \parallel traces(TeachingAssistant) \parallel traces(Professor)\}$

In scenario 2, *Student* sends *squestion* to *TeachingAssistant*, and then *TeachingAssistant* sends *tques- tion* to *Professor*. After receiving *tquestion*, *Professor* sends *tanswer* to *TeachingAssistant*, and then *TeachingAssistant* sends *tanswer* to *Student*.

The event traces modeling the communications in both scenarios 1 and 2 are formalized below.

$traces(Student \parallel TeachingAssistant \parallel Professor)$

$=\{\langle\rangle, \langle squestion\rangle, \langle squestion, tanswer\rangle^\frown s \mid$

$s \in traces(Student) \parallel traces(TeachingAssistant) \parallel traces(Professor)\}$

$\cup \{\langle\rangle, \langle squestion\rangle, \langle squestion, tquestion\rangle, \langle squestion, tquestion, panswer\rangle,$

$\langle squestion, tquestion, panswer, tanswer\rangle^\frown s \mid$

$s \in traces(Student) \parallel traces(TeachingAssistant) \parallel traces(Professor)\}$

According to the generated traces of events, the system in implementation will first behave as $\{\langle\rangle, \langle squestion\rangle, \langle squestion, tanswer\rangle\}$ or $\{\langle\rangle, \langle squestion\rangle, \langle squestion, tquestion\rangle, \langle squestion, tquestion, panswer\rangle, \langle squestion, tquestion, panswer, tanswer\rangle\}$, and then behave as $traces(Student)$ $\| traces(TeachingAssistant) \| traces(Professor)$.

Illustration of Step 4: Build Categorical Models of Traces from Design

The aim of this step is to construct categories for modeling progress of communications in the design. The progress of communications can be indicated by traces of events. In Chapter 3, the categories of traces in proposition 2 are provided as follows.

- **Category of Traces**: Each object is a set of traces to indicate a process. A morphism $traces(A) \rightarrow traces(B)$ means traces of process A evolves to traces of process B, where $traces(A) \subseteq traces(B)$.

Proof of constructing category of traces is provided in Chapter 3.

Proposition 4. DEvents is a type of category of traces. It captures the designed behaviors of the system based on traces of events extracted from the design. In **DEvents**, each object represents a set of traces of communications the system designed; each morphism models $\subseteq$ relationship between sets of traces to indicate the progress of the system; and each identity represents the set of traces $\subseteq$ itself. The category **DEvents** is a type of category of traces.

Fig. 4.5 illustrates part of **DEvents** category with the first few traces of unbounded sequences.

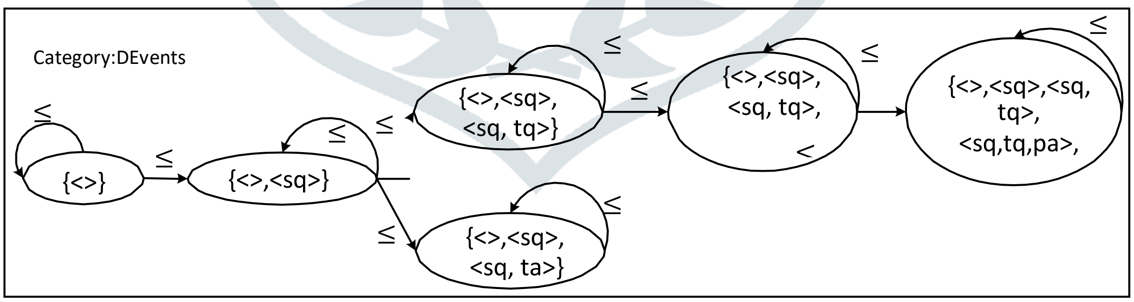

Figure 4.5: Category of Traces from the Design

Proof.

Objects: Each object is a set of traces of events, such as $\{\langle\rangle\}$, $\{\langle\rangle, \langle sq\rangle\}$, and $\{\langle\rangle, \langle sq\rangle, \langle sq, tq\rangle\}$.

Morphisms: Let $traces(A)$ and $traces(B)$ be objects. If $traces(A) \subseteq traces(B)$, there is a morphism $traces(A) \rightarrow traces(B)$.

Identities: For each object, $traces(A)$, there is an identity $traces(A) \rightarrow traces(A)$, which indicates $traces(A) \subseteq traces(A)$.

Composition: Given any morphisms $morph_{A,B} : traces(A) \rightarrow traces(B)$ and $morph_{B,C} : traces(B) \rightarrow traces(C)$, with codomain of $morph_{A,B}$ = domain of $morph_{B,C}$, there is $traces(A) \subseteq traces(B) \subseteq traces(C)$. Thus, there is a composition morphism: $morph_{B,C} \circ morph_{A,B} : traces(A) \rightarrow traces(C)$, which means $traces(A) \subseteq traces(C)$.

Associativity: For all morphisms $morph_{A,B} : traces(A) \rightarrow traces(B)$, $morph_{B,C} : traces(B) \rightarrow traces(C)$ and $morph_{C,D} : traces(C) \rightarrow traces(D)$, with codomain of $morph_{A,B}$ = domain of $morph_{B,C}$ and codomain $morph_{B,C}$ = domain of $morph_{C,D}$, there is $traces(A) \subseteq traces(B) \subseteq traces(C) \subseteq trace(D)$. Thus, there are $morph_{C,D} \circ (morph_{B,C} \circ morph_{A,B}) = morph_{C,D} \circ (traces(A) \rightarrow trace(C)) = traces(A) \rightarrow traces(D)$, and $(morph_{C,D} \circ morph_{B,C}) \circ morph_{A,B} = (traces(B) \rightarrow traces(D)) \circ morph_{A,B} = traces(A) \rightarrow traces(D)$. So, $morph_{C,D} \circ (morph_{B,C} \circ morph_{A,B}) = (morph_{C,D} \circ morph_{B,C}) \circ morph_{A,B}$ □

Illustration of Step 5: Build Categorical Models of Trace from Abstraction of Implementation

The aim of this step is to construct categories for communications in the abstraction of imple- mentation. The progress of communications can be indicated by traces of events. In Chapter 3, the categories of traces in proposition 2 is provided as follows.

- **Category of Traces**: Each object is a set of traces to indicate a process. A morphism $traces(A) \rightarrow traces(B)$ means traces of process A evolves to traces of process B, where $traces(A) \subseteq traces(B)$.

Proof of constructing category of traces is provided in Chapter 3.

Proposition 5. IEvents is a type of category of traces. It captures the implemented behaviors of the system based on traces of events extracted from the abstraction in section 5.4.3. In **IEvents**, each object represents a trace of events of the system implemented; each morphism models $\subseteq$ relationship

between sets of traces to indicate the progress of the system; and each identity represents the set of traces $\subseteq$ itself.

Fig. 4.6 illustrates part of **IEvents** category with the first few traces of unbounded sequences.

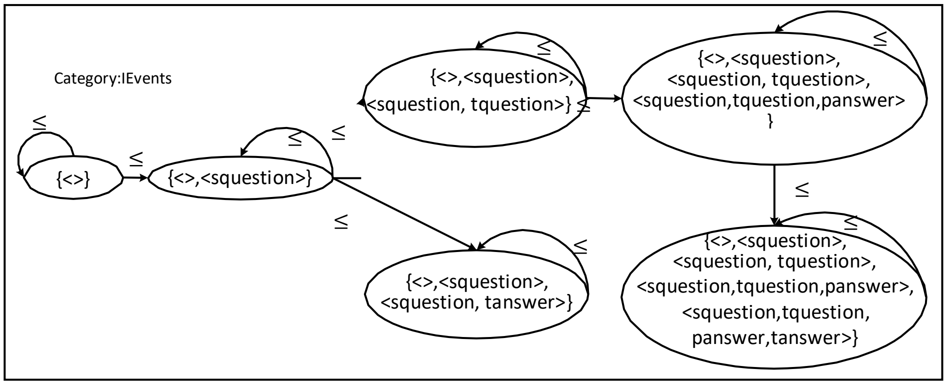

Figure 4.6: Category of Traces from the Implementation

Proof.

Objects: Each object is a set of traces of events, such as $\{\langle\rangle\}$ and $\{\langle\rangle, \langle squestion\rangle\}$.

Morphisms: Let $traces(A)$ and $traces(B)$ be objects. If $traces(A) \subseteq traces(B)$, there is a morphism $traces(A) \rightarrow traces(B)$.

Identities: For each object, $traces(A)$, there is an identity $traces(A) \rightarrow traces(A)$, which indicates $traces(A) \subseteq traces(A)$.

Composition: Given any morphisms $morph_{A,B} : traces(A) \rightarrow traces(B)$ and $morph_{B,C} : traces(B) \rightarrow traces(C)$, with codomain of $morph_{A,B}$ = domain of $morph_{B,C}$, there is $traces(A) \subseteq traces(B) \subseteq traces(C)$. Thus, there is a composition morphism: $morph_{B,C} \circ morph_{A,B} : traces(A) \rightarrow traces(C)$, which means $traces(A) \subseteq traces(C)$.

Associativity: For all morphisms $morph_{A,B} : traces(A) \rightarrow traces(B)$, $morph_{B,C} : traces(B) \rightarrow traces(C)$ and $morph_{C,D} : traces(C) \rightarrow traces(D)$, with codomain of $morph_{A,B}$ = domain of $morph_{B,C}$ and codomain $morph_{B,C}$ = domain of $morph_{C,D}$, there is $traces(A) \subseteq traces(B) \subseteq traces(C) \subseteq trace(D)$. Thus, there are $morph_{C,D} \circ (morph_{B,C} \circ morph_{A,B}) = morph_{C,D} \circ (traces(A) \rightarrow trace(C)) = traces(A) \rightarrow traces(D)$, and $(morph_{C,D} \circ morph_{B,C}) \circ morph_{A,B} =$

$(traces(B) \rightarrow traces(D)) \circ morph_{A,B} = traces(A) \rightarrow traces(D)$. So, $morph_{C,D} \circ (morph_{B,C} \circ morph_{A,B}) = (morph_{C,D} \circ morph_{B,C}) \circ morph_{A,B}$ □

Illustration of Step 6: Construct Functors from Categories of Design to Cate- gories of Abstraction of Implementation

The aim of this step is to verify consistency between design and implementation by constructing categories and functors. According to Chapter 3, consistency between the design and the implemen- tation is defined as follows.

Consistency of Communications with Traces: Given a sequence of sets of traces in the de- sign representing the progress of the system, $DTraces : \{\langle\rangle\} \rightarrow \{\langle\rangle, \langle devent_1 \rangle\} \rightarrow \cdots \rightarrow \{\langle\rangle, \langle devent_1 \rangle, \cdots, \langle devent_1, \ldots, devent_n \rangle\}$, and a sequence of traces in the implementation rep- resenting the progress of the system, $ITraces : \{\langle\rangle\} \rightarrow \{\langle\rangle, \langle ievent_1 \rangle\} \rightarrow \cdots \rightarrow \{\langle\rangle, \langle ievent_1 \rangle, \cdots, \langle ievent_1, \ldots, ievent_n \rangle\}$. If there exists a mapping from *DTraces* to *ITtraces* with sequence pre- served, which can map $\{\langle\rangle, \langle devent_1 \rangle, \cdots, \langle devent_1, \ldots, devent_i \rangle\}$ to $\{\langle\rangle, \langle ievent_1 \rangle, \cdots, \langle ievent_1, \ldots, ievent_i \rangle\}$, and $\{\langle\rangle, \langle devent_1 \rangle, \cdots, \langle devent_1, \ldots, devent_i, devent_{i+1} \rangle\}$ to $\{\langle\rangle, \langle ievent_1 \rangle, \cdots, \langle ievent_1, \ldots, ievent_i, ievent_{i+1} \rangle\}$, then *ITraces* is consistent with *DTraces*. If all sequences in the design have corresponding mapping sequences in the implementation, the communications in the implementation are consistent with the communications in the design.

To verify consistency of communications with traces between design and implementation, the construction of a functor can be used [55, 56, 57, 58]. If there exists a functor that maps the category of traces from design to the category of traces from implementation, the implementation is consis- tent with the design. Otherwise, the implementation is inconsistent with the design. According to Chapter 3, the functor can be constructed with the following approach.

- For each object, *ocd*, in design, there must be a corresponding object, *oci*, in implementation, such that *ocd* can be mapped to *oci* when each trace in *ocd* has the same trace in *oci*.
- For each morphism $md : ocd1 \rightarrow ocd2$ in design, there must be a corresponding morphism $mi : oci1 \rightarrow oci2$ in implementation, such that *md* can be mapped to *mi* when *ocd1* and *ocd2* can be mapped to *oci1* and *oci2* respectively.

Based on the analysis of categories **DEvents** and **IEvents**,the consistency between the design and the implementation is verified by constructing a functor **DToI**: **DEvents** → **IEvents**. This functor maps objects and morphisms of **DEvents** to the corresponding objects and morphisms of **IEvents** as follows:

- Objects Mapping: an object *od* of **DEvents** maps to an object *oi* of **IEvents**, when the trace in *od* matches the trace in *oi*. For example, $\{\langle\rangle, \langle squestion\rangle\}$ in **IEvents** represents an event that *Student* sends a *question* to *TeachingAssistant* in the implementation, and $\{\langle\rangle, \langle sq\ \rangle\}$ in **DEvents** represents an event that *Student* sends a *question* to *TeachingAssistant* in the design. Thus, $\{\langle\rangle, \langle sq\ \rangle\}$ matches $\{\langle\rangle, \langle squestion\rangle\}$.

- Morphisms Mapping: a morphism $md : od_1 \rightarrow\ od_2$ of **DEvents** maps to a morphism $mi : oi_1 \rightarrow\ oi_2$ of **IEvents**, when od_1 and od_2 match oi_1 and oi_2 respectively, and $\rightarrow$ from od_1 to od_2 matches $\rightarrow$ from oi_1 to oi_2. For example, $\{\langle\rangle, \langle sq\rangle\} \rightarrow\ \{\langle\rangle, \langle sq\ \rangle, \langle sq, ta\rangle\}$ maps to $\{\langle\rangle, \langle squestion\rangle\} \rightarrow\ \{\langle\rangle, \langle squestion\rangle, \langle squestion, tanswer\ \rangle\}$.

- Identities Mapping: By following the objects mapping and morphisms mapping, identity mapping is preserved from **DEvents** to **IEvents**.

- Composition Morphisms Mapping: By following the objects mapping and morphisms map- ping, compositions of morphisms mapping are preserved from **DEvents** to **IEvents**.

Fig. 4.7 shows that **DToI**: **DEvents** → **IEvents** is a functor.

A successful construction of the functor **DToI** indicates that the implementation and the design are consistent.

Summary

In this chapter, the categorical framework is used to verify consistency of communications with traces between design and implementation. This framework used traces, category theory and ab- straction of implementation, and is illustrated by a running system with processes *Student*, *Teachin- gAssistant* and *Professor* executing in parallel. In doing so, the design of the system is modeled and

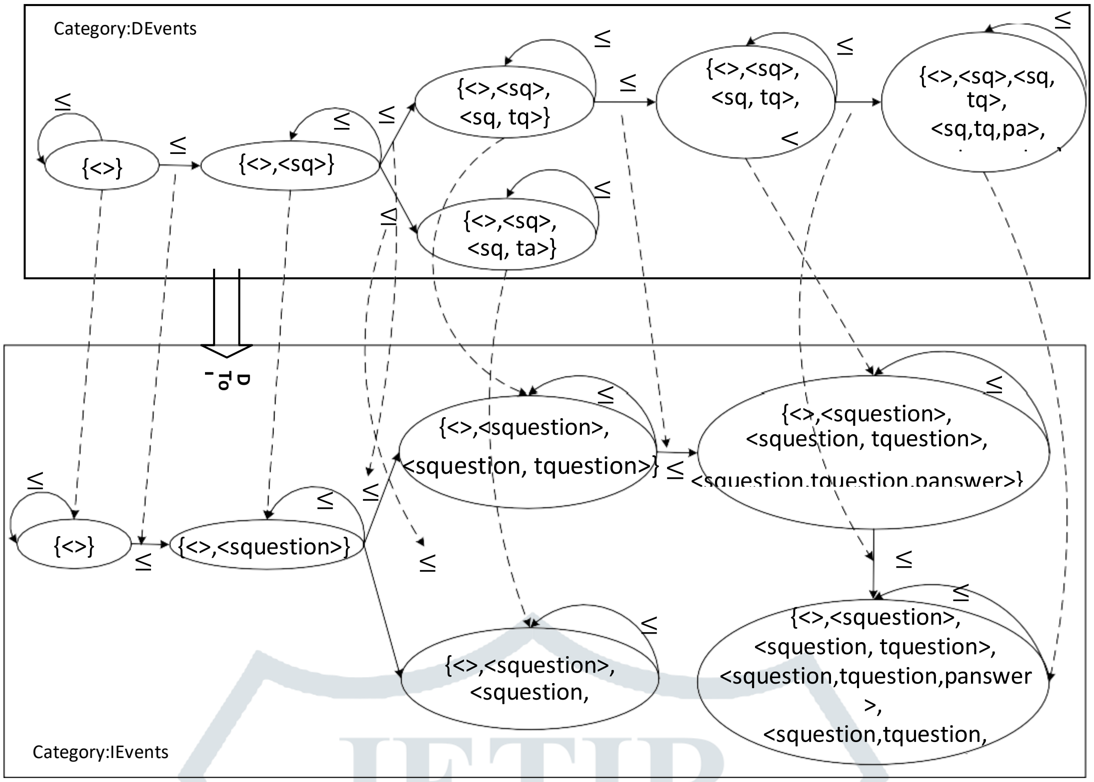

Figure 4.7: A Functor from the Category of Traces in Design to the Category of Traces in the Abstraction of Implementation

analyzed by CSP, the implementation of the system is created by Erasmus, traces of events of the implementation are analyzed based on abstraction, categories of traces of events from the design and implementation are created, and, by constructing a functor, the consistency between the design and the implementation is verified.

In the next chapter, we introduce how to use the categorical framework to verify consistency of communications failures between design and implementation.

Chapter 5

Verifying Communications with Failures

Introduction

A process can be modeled in terms of failures that can represent both liveness and safety of the process. In this chapter, by using the categorical framework, we can verify consistency of com- munications with failures between design and implementation. Section 5.2 briefs the contributions in verifying communications with failures. Section 5.3 introduces the categorical framework for verifying communications with failures between design and implementation. Section 5.4 gives an overview of a running example with three different implementation scenarios to illustrate the appli- cation of the framework for verification with failures. Section 5.5 summarizes this chapter.

Contributions

Several contributions in verifying communications with failures are introduced as follows:

- The framework for verification with failures is proposed.
- Category theory is used to model communications with failures in design and implementation.
- Functors are used to verify consistency of communications with failures between design and implementation.

The Framework for Verification with Failures

As stated in Chapter 3, we apply the framework to model and analyze the consistency of com- munications with failures. Fig. 5.1 depicts the process of communication verification with failures in the categorical framework. The steps of the verification process are outlined next.

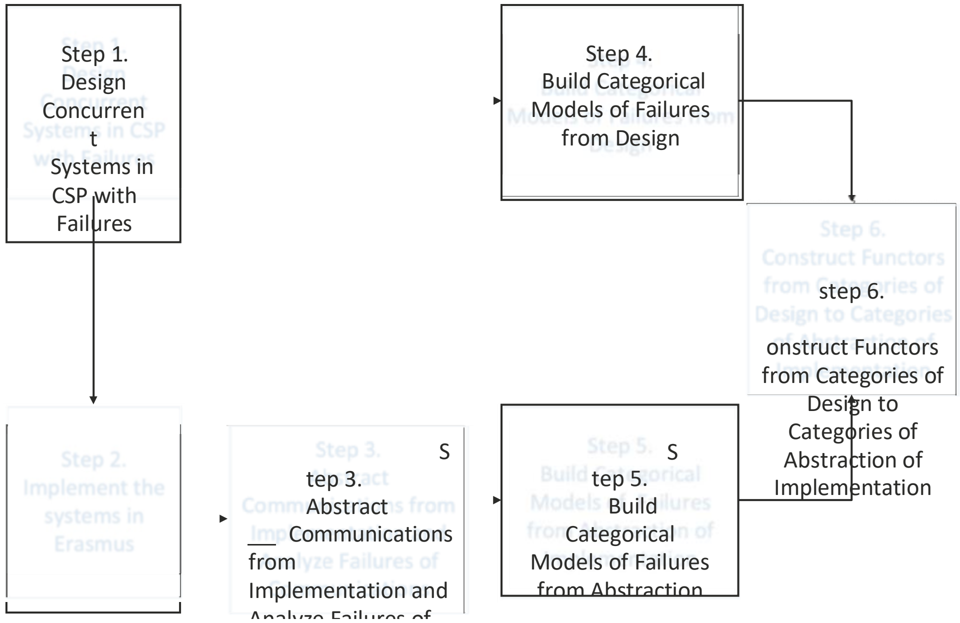

Figure 5.1: The Categorical Framework for Verification with Failures

Step 1. Design Concurrent Systems in CSP with Failures: In this step, we need to design concurrent systems in CSP, and then analyze failures of processes together with communications. This step is to achieve research objective OBJ1.

Step 2. Implement the Systems in Erasmus: In this step, we need to implement the concurrent systems in Erasmus by refining the design in step 1. This step is to achieve research objective OBJ2. Step 3. Abstract Communications from Implementation and Analyze Failures of Communica- tions: In this step, we need to abstract processes and communications out of the implementation in step 2, and then analyze failures of abstract processes as well as communications. This step is to

achieve research objective OBJ3.

Step 4. Build Categorical Models of Failures from Design: In this step, we need to construct categorical models for the design in step 1 with preserving structures of communications. This step

is to achieve research objective OBJ4.

Step 5. Build Categorical Models of Failures from Abstraction of Implementation: In this step, we need to construct categorical models for the abstraction of implementation in step 3 with preserving structures of communications. This step is to achieve research objective OBJ5.

Step 6. Construct Functors from Categories of Design to Categories of Abstraction of Imple- mentation: In this step, we need to construct functors to verify the categorical models of the design in step 4 and the categorical model of abstraction of implementation in step 5. This step is to achieve research objective OBJ6.

To illustrate the process of verification with failures, the workflow of the framework are de- scribed by a running example in the following sections.

Illustration of a Running Example

To illustrate the categorical framework, a client/server example is developed. In the example, the concurrent system consists of two processes *server* and *client*.

- The *server* can provide two types of service, *serviceA* and *serviceB*. The *client* can request *serviceA* and *serviceB*.
- In the beginning, the *client* lets the *server* know the type of service it requests.
- Then, the *client* sends the information related to the requested service to the *server*.
- At last, the *client* receives the corresponding results from the *server*.
- The *client* can repeatedly request service from *server*.

The graphical representation of this example is given in Fig. 5.2.

According to the software development process, we develop the design in CSP based on the requirements specification of the example, then we refine the design into the implementation in Erasmus. In order to demonstrate the application of the framework can indicate whether commu- nications of process are consistent or inconsistent between design and and implementation. In the implementation stage, we develop three different scenarios.

Figure 5.2: The Client/Server Example

- In the first scenario, the *server* offers three types of services that are *serviceA*, *serviceB* and *serviceC*.
- In the second scenario, the *server* offers only one type of services that is *serviceA*.
- In the third scenario, the *server* offers *serviceA* and *serviceB* as designed.

With the application of the categorical framework for verification with failures to the example, the consistency of client/server communications between the design and the implementation can be verified automatically.

Illustration of Step 1: Design Concurrent Systems in CSP with Failures

The aim of this step is to design and analyze the processes and the concurrent system in CSP based on the textual description of the system requirements.

Step 1.a: Model the Conceptual Design

As CSP can model and specify processes in concurrent system, for this example, the design of the above described system is specified as follows:

$client = requestA \rightarrow infoA \rightarrow resultA \rightarrow client \ \mathrm{H} \ requestB \rightarrow infoB \rightarrow resultB \rightarrow$
$client\ server = requestA \rightarrow infoA \rightarrow resultA \rightarrow Server \ \mathrm{Q} \ requestB \rightarrow infoB \rightarrow resultB$
$\rightarrow Server$

In this design, *client* represents the process client; *server* represents the process server; *reques- tA*,*infoA*,*resultA*, *requestB*,*infoB*,*resultB* are events communicated between process client and pro- cess server; → denotes the "occurs before" relation between events; H means the nondeterministic
choices made by the process itself; and Q stands for the deterministic choices based on the event
from the environment.

Step 1.b: Generate and Analyze Failures

To analyze the behaviors of a concurrent system, we need to analyze failures. Failures of a process is defined as a relation (set of pairs)

$$failures(P) = \{(s, X) \mid s \in traces(P) \wedge X \in refusals(P/s)\}$$

If (s, X) is a failure of P, this means that P can engage in the sequence of events recorded by s, and then, refuse to do anything more, in spite of the fact that its environment is prepared to engage in any of the events of X [4].

To generate and analyze failures of processes in CSP, according to Chapter 3, several rules defined in CSP [4, 5] are used in this research.

$failures(c \rightarrow P) = \{(\langle\rangle, X) | c \notin X\} \cup \{(\langle c\rangle^\frown s, X) | (s, X) \in failures(P)\}$

$failures(P ; Q) = \{(s, X) | s \in A^* \wedge (s, X \cup \{C\}) \in failures(P)\}$

$\cup \{(s^\frown t, X) | s^\frown\langle C\rangle \in (traces)(P) \wedge (t, X) \in failures(Q)\}$ $failures(P \text{ Q } Q) = \{(\langle s\rangle, X) | (\langle\rangle, X) \in failures(P) \cap failures(Q)$

$\vee (s \neq \langle\rangle \wedge (s, X) \in failures(P) \cup failures(Q))\}$ $failures(P$

$\text{H } Q) = failures(P) \cup failures(Q)$

$failures(P \parallel Q) = \{(s, X \cup Y) | s \in A^* \wedge (s, X) \in failures(P) \wedge (s, Y) \in failures(Q)\}$

Model Individual Processes with Failures

For the client/server example, according to the above mentioned rules of CSP, failures of pro- cesses *client* and *server* can be generated and analyzed as follows:

$failures(client) =$

$\{\{(\langle\rangle, X) | X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\}\},$

$\{(\langle requestA\rangle, X) | X \subseteq \{requestA, resultA, requestB, infoB, resultB\}\},$

$\{(\langle requestB\rangle, X) | X \subseteq \{requestA, infoA, resultA, requestB, resultB\}\},$

$\{(\langle requestA, infoA\rangle, X) | X \subseteq \{requestA, infoA, requestB, infoB, resultB\}\},$

$\{(\langle requestB, infoB\rangle, X) | X \subseteq \{requestA, infoA, resultA, requestB, infoB\}\},$

$\{(\langle requestA, infoA, resultA\rangle, X) | X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\}\},$

$\{(\langle requestB, infoB, resultB\rangle, X) | X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\}\},$

$\ldots\}$

$failures(server) =$

$\{\{(\langle\rangle, X) | X \subseteq \{infoA, resultA, infoB, resultB\}\},$

$\{(\langle requestA\rangle, X) | X \subseteq \{requestA, resultA, requestB, infoB, resultB\}\},$

$\{(\langle requestB\rangle, X) | X \subseteq \{requestA, infoA, resultA, requestB, resultB\}\},$

$\{(\langle requestA, infoA\rangle, X) | X \subseteq \{requestA, infoA, requestB, infoB, resultB\}\},$

$\{(\langle requestB, infoB\rangle, X) | X \subseteq \{requestA, infoA, resultA, requestB, infoB\}\},$

$\{(\langle requestA, infoA, resultA\rangle, X) | X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\}\},$

$\{(\langle requestB, infoB, resultB\rangle, X) | X \subseteq \{infoA, resultA, infoB, resultB\}\},$

$\ldots\}$

In this listing of failures, *failures*(P) stands for generating a set of all possible failures of process P ; X in ($trace, X$) is a refusal of the *trace*; $\langle event_1, \cdots, event_n\rangle$ indicates a specific trace of events.

Model Communications between Processes with Failures

When processes *client* and *server* work in parallel as a system, CSP operator "$\parallel$" models com- munication between processes. According to CSP, if there is a communication between two process- es, there must be an event that occurs in both processes simultaneously. Failures of communications between *client* and *server* can be generated, analyzed and represented as follows:

$failures(client \parallel server\ ) =$

$\{\{(\langle\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\ \}\},$

$\{(\langle requestA\rangle, X\ ) \mid X \subseteq \{requestA, resultA, requestB, infoB, resultB\ \}\}$

$\{(\langle requestB\ \rangle, X\ ) \mid X \subseteq \{requestA, infoA, resultA, requestB, resultB\ \}\}$

$\{(\langle requestA, infoA\rangle, X\ ) \mid X \subseteq \{requestA, infoA, requestB, infoB, resultB\ \}\}$

$\{(\langle requestB, infoB\ \rangle, X\ ) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB\ \}\}$

$\{(\langle requestA, infoA, resultA\rangle, X\ ) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\ \}\}$

$\{(\langle requestB, infoB, resultB\ \rangle, X\ ) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\ \}\}$

$\ldots\}$

Illustration of Step 2: Implement the Systems in Erasmus

The aim of this step is to implement the processes and the concurrent system in Erasmus based on the design. As there are three different scenarios, each will be implemented in Erasmus in the following sections.

Implement Scenario 1

In this scenario, process *server* provides *serviceA*, *serviceB* and *serviceC*, and process *client* requests all services from *server*. The Erasmus code for the implementation is as follows.

```
match = protocol { requestA|infoA|ˆresultA
|requestB|infoB|ˆresultB
                                    |requestC|infoC|ˆresultC}
  //message without ˆ is a request.
  //message with ˆ in front is a reply.
  //all messages in communications
  //requests and info are sent by client
  //results are sent by server

  server = process p: +match{ //process server loop select{
                ||p.requestA; p.infoA; p.resultA; //serviceA
                ||p.requestB; p.infoB; p.resultB; //serviceB
                ||p.requestC; p.infoC; p.resultC;} //serviceC
  }

  client = process e: -match{ //process client loop case{
                ||e.requestA; e.infoA; e.resultA; //serviceA
                ||e.requestB; e.infoB; e.resultB; //serviceB
                ||e.requestC; e.infoC; e.resultC;} //serviceC
  }
  //encapsulate processes
  Main = cell{ m: Channel match; server(m); client(m); }
```

Implement Scenario 2

In this scenario, process *server* is implemented to provide only one type of service, and process *client* is implemented to request the service from *server*. The Erasmus code for the implementation is as follows.

```
match = protocol {requestA |infoA |ˆresultA}
//message without ˆ is a request.
//message with ˆ in front is a reply.
//all messages in communications
//requests and info are sent by client
```

```
//results are sent by server

server = process p: +match{ //process server loop{
        p.requestA; p.infoA; p.resultA; } //serviceA
}

client = process e: -match{ //process client loop{
        e.requestA; e.infoA; e.resultA; } //serviceA
}

//encapsulate processes
Main = cell{m: Channel match; server(m); client(m);}
```

Implement Scenario 3

In this scenario, process *server* provides *serviceA* and *serviceB*, and process *client* requests both services from Server. The services implemented in this scenario are as same as the services in the design. The Erasmus code for the implementation is as follows.

```
match = protocol { requestA|infoA|ˆresultA
                   |requestB|infoB|ˆresultB}
//message without ˆ is a request.
//message with ˆ in front is a reply.
//all messages in communications
//requests and info are sent by client
//results are sent by server

server = process p: +match{ //process server loop select{
        ||p.requestA; p.infoA; p.resultA; //serviceA
        ||p.requestB; p.infoB; p.resultB;} //serviceB
}
client = process e: -match{ loop case{
        ||e.requestA; e.infoA; e.resultA; //serviceA
        ||e.requestB; e.infoB; e.resultB;} //serviceB
}
//encapsulate processes
Main = cell{m: Channel match; server(m); client(m);}
```

Illustration of Step 3: Abstract Communications from Implementation and Analyze Failures of Communications

Since the interest in this thesis is in analyzing the behaviors of the system based on failures, an abstraction is created for extracting the code pertaining to generate communications with failures. The aim of this step is to use Galois connection to abstract processes and communications from the implementation, and analyze processes and communications with failures in Erasmus.

Step 3.a.1: Abstract the Implementation of Scenario 1

According to the abstraction rules in Chapter 3, the abstraction of implementation contains loops, deterministic choices, nondeterministic choices, sending and receiving messages through ports. The abstraction of the implementation of scenario 1 is provided as follows.

```
server =
        loop{select{
p.requestA; p.infoA; p.resultA;
|p.requestB; p.infoB; p.resultB;
|p.requestC; p.infoC; p.resultC}
   }

client =
        loop{case{
e.requestA; e.infoA; e.resultA;
|e.requestB; e.infoB; e.resultB;
|e.requestC; e.infoC; e.resultC}
   }
```

In the abstraction of the implementation, **loop** can be defined by recursion; **select** together with | represents deterministic choices; **case** together with | represents nondeterministic choices; the notation *port.message*(for example *p.requestA*) represents *message*(*requestA*) that occurs on *port*(*p*); and the symbol ";" is the delimiter to indicate the "occurs before" relation between messages.

In this scenario, implementation is considered as concrete domain, and abstraction is considered as abstract domain. The relationships "execute before or simultaneously" between statements in abstraction are maintained in implementation, and vice versa. The details of mappings for this scenario are shown in Fig. 5.3:

Step 3.a.2: Abstract the Implementation of Scenario 2

According to the abstraction rules in Chapter 3, the abstraction of implementation contains loops, deterministic choices, nondeterministic choices, sending and receiving messages through ports. The abstraction of the implementation of scenario 2 is provided as follows.

```
server =
    loop{
        p.requestA; p.infoA; p.resultA
}
client =
    loop{
        e.requestA; e.infoA; e.resultA
}
```

In this scenario, implementation is considered as concrete domain, and abstraction is considered as abstract domain. The relationships "execute before or simultaneously" between statements in abstraction are maintained in implementation, and vice versa. The details of mappings for this scenario are shown in Fig. 5.4:

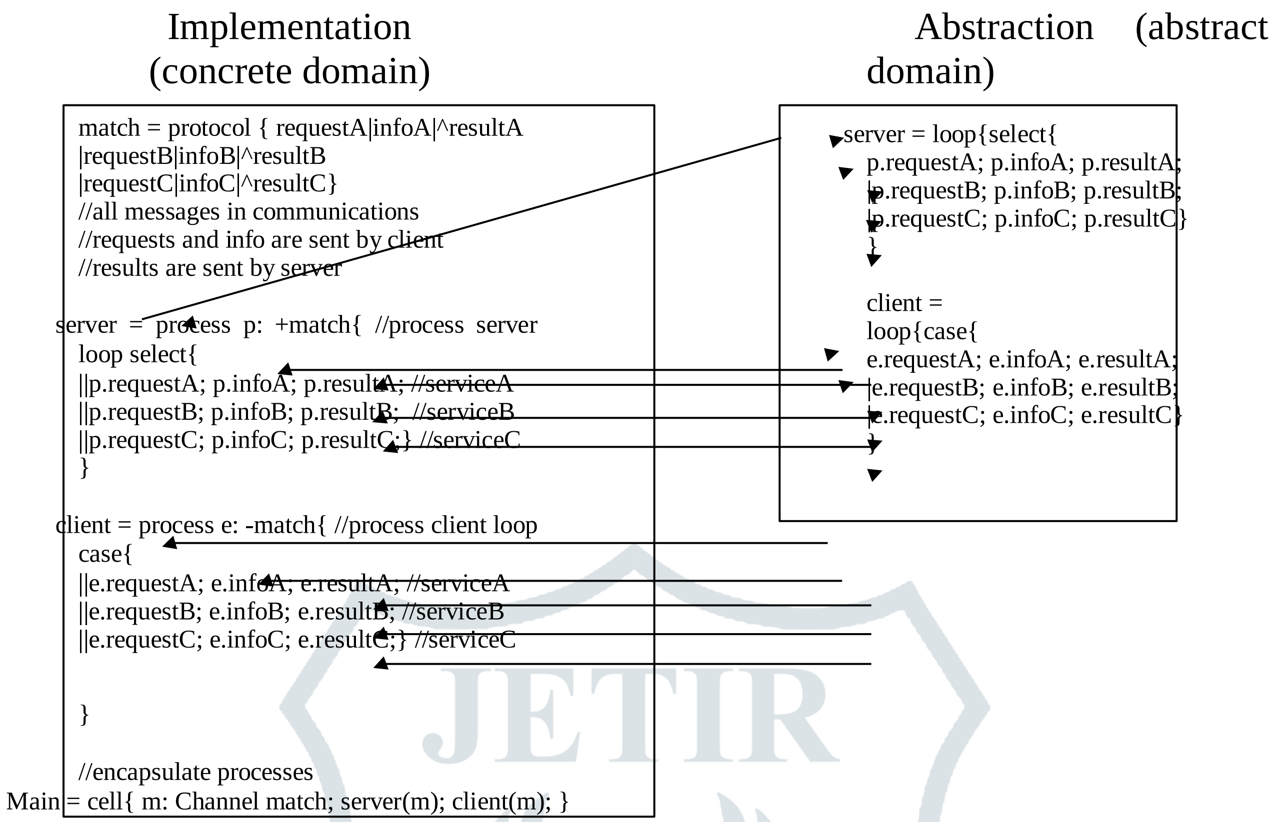

Figure 5.3: Mappings Between Implementation and Abstraction of Scenario 1 of the Client/Server Example

Implementation (concrete domain) Abstraction (abstract domain)

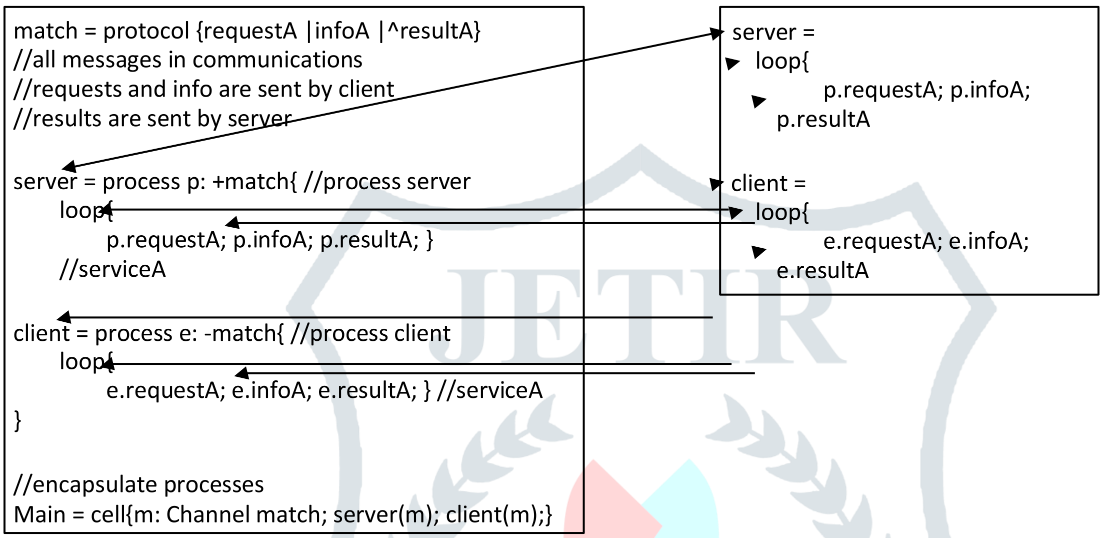

Figure 5.4: Mappings Between Implementation and Abstraction of Scenario 2 of the Client/Server Example

Step 3.a.3: Abstract the Implementation of Scenario 3

According to the abstraction rules in Chapter 3, the abstraction of implementation contains loops, deterministic choices, nondeterministic choices, sending and receiving messages through ports. The abstraction of the implementation of scenario 3 is provided as follows.

```
server =
     loop{select{
            p.requestA; p.infoA; p.resultA
          |p.requestB; p.infoB; p.resultB }
}

client =
     loop{case{
            e.requestA; e.infoA; e.resultA
          |e.requestB; e.infoB; e.resultB }
}
```

In this scenario, implementation is considered as concrete domain, and abstraction is considered as abstract domain. The relationships “execute before or simultaneously” between statements in abstraction are maintained in implementation, and vice versa. The details of mappings for this scenario are shown in Fig. 5.5:

Step 3.b: Generate and Analyze Failures

A process in Erasmus usually has one or more ports for communications, which differs from the process in CSP. A set of all messages a port can send or receive is considered as the $alphabet_{port}$. A set of messages of all ports of a process is deemed as the $alphabet_{process}$=\{$alphabet_{port1} \cup \ldots \cup alphabet_{port_n}$\}. To model implementation, a process can be modeled by using ports, where a port can be modeled as ($alphabet_{port}$, $failures_{port}$).

Although the syntax of Erasmus is different from CSP, the semantics of Erasmus is analogous to CSP. Some notions and rules that model failures in CSP can be also used to model failures in Erasmus with preserving the same syntax and semantics, which includes $\bar{}$, $\cup$, $\langle\rangle$, H and Q.

Implementation (concrete domain)

```
match = protocol { requestA|infoA|^resultA
                          |requestB|infoB|^resul
          tB}
//all messages in communications
//requests and info are sent by client

server = process p: +match{ //process server
     loop select{
          ||p.requestA; p.infoA; p.resultA;
     //serviceA
          ||p.requestB; p.infoB; p.resultB;}

client = process e: -match{
     loop case{
          ||e.requestA; e.infoA; e.resultA; //serviceA
          ||e.requestB; e.infoB; e.resultB;} //serviceB
}

//encapsulate processes
Main = cell{m: Channel match; server(m); client(m);}
```

Abstraction (abstract domain)

```
server =
     loop{select{
          p.requestA; p.infoA;
     p.resultA
          |p.requestB; p.infoB;

client =
     loop{case{
          e.requestA; e.infoA;
     e.resultA
          |e.requestB; e.infoB;
```

Figure 5.5: Mappings Between Implementation and Abstraction of Scenario 3 of the Client/Server Example

To generate and analyze the traces of processes in Erasmus, according to Chapter 3, the following rules are proposed in this research.

(1). Let P be a process, let p be a port of P, and let m be the first message that will be sent/received through prot p. The message can be represented $P.p.m$. $P.p.m$ is a simple statement. If port p is unique in the system, $P.p.m$ can be abbreviated as $p.m$. The failures of port p of process P for sending/receiving message m are $failures(P.p.m) = \{(\langle\rangle, X) | X \subseteq (alphabet(p) - m)\}$. It means any event occurs on port p other than message m, p stops working.

(2). Let C_1 and C_2 be two statements , and let C_1 execute before C_2. There is $C_1; C_2$, which is a compound statement with the failures $failures(C_1; C_2) = \{(s, X) \mid (s, X) \in failures(C_1)\} \cup \{(s^\frown t, Y) \mid s^\frown\langle C\rangle \in traces(C_1) \wedge (t, Y) \in failures(C_2)\}$. It means that the failures $failures(C_1; C_2)$ become $failures(C_1)$ first, as C_1 executes before C_2. After C_1 accomplishing its execution with trace s successfully, the failures $failures(C_1; C_2)$ depend on $failures(C_2)$.

(3). Let C be a statement iterating n times in a loop, and let C^i represent the ith iteration of a loop of C. There is $loop\{C\} = \{C^1; C^2 \ldots C^{n-1}; C^n\}$, which is a compound statement with the failures $failures(loop\{C\}) = \{(s, X) \mid (s, X) \in failures(C)\} \cup \{(s^{1\frown}s, X) \mid s^{1\frown}\langle C\rangle \in traces(C) \wedge (s, X) \in failures(C)\} \cup \ldots \cup \{(s^{1\frown}s^{2\frown} \ldots {}^\frown s^{n-1\frown}s^n, X) | s^{i\frown}\langle C\rangle \in traces(C) \wedge 1 \leq i \leq n - 1 \wedge (s, X) \in (failures(C))\}$. It means that if C iterates once, $failures(loop\{C\})$ become $failures(C)$; if C iterates twice, and if the execution of the first iteration is accomplished successfully with trace s^1, $failures(loop\{C\})$ depends on $failures(C)$ in the second iteration; if C iterates n times, and if the execution from $1st$ iteration to $(n - 1)th$ iteration successfully with trace $s^{1\frown}s^{2\frown} \ldots {}^\frown s^{n-1}$, $failures(loop\{C\})$ depend on $failures(C)$ in the nth iteration.

(4). Let C_i be a statement where $1 \leq i \leq n$, and let $case$ represent nondeterministic choices. There is $case\{C_1 | \ldots | C_n\}$, which is a compound statement with $failures(case\{C_1 | \ldots | C_n\}) = \{(s, X) | (s, X) \in failures(C_1) \cup \ldots \cup failures(C_n)\}$. It means that $failures(case\{C_1 | \ldots | C_n\})$ depends on one of $failures(C_i)$ where $1 \leq i \leq n$.

(5). Let C_i be a statement where $1 \leq i \leq n$, and let $select$ represent deterministic choices. There is $select\ \{C_1 | \ldots | C_n\}$, which is a compound statement with the failures $failures(itselect\{C_1 | \ldots | C_n\}) = \{(s, X) | (s = \langle\rangle \wedge (s, X) \in failures(C_1) \cap \ldots \cap failures(C_n)) \vee (s \quad \langle\rangle \wedge (s, X) \in failures(C_1) \cup \ldots \cup failures(C_n))\}$. It means that if statements C_i wait for the occurrence of the first message, $failures(select\{C_1 | \ldots | C_n\})$ would become $failures(C_1) \cap \ldots \cap failures(C_n)$. When the trace s occurs, it indicates one of C_i executes, so $failures(select\{C_1 | \ldots | C_n\})$ would become $failures(C_1) \cup \ldots \cup failures(C_n)$.

(6). Let C_1 be a statement from a process, let C_2 be a statement from another process, and let C_1 and C_2 be able to communicate with each other. There is $C_1 \parallel C_2$, which is a compound statement with $failures(C_1 \parallel C_2) = \{(s, X \cup Y) | ((s, X) \in failures(C_1) \wedge (s, Y) \in failures(C_2))\}$. In Erasmus, two ports can communicate only when the same message is sent by a port and received by another port simultaneously. If there is a failure of $C_1 \parallel C_2$, the failure would be from either $failures(C_1)$ or $failures(C_2)$.

Step 3.b.1: Generate and Analyze Failures of Scenario 1

In this implementation scenario, process *client* has only one port *e*, and process *server* has only one port *p*. Thus, *client* can be represented as {*alphabet*(*e*), *failures*(*e*)}, and *server* can be represented as {*alphabet*(*p*), *failures*(*p*)}.

As we are interested in communications between processes, in the abstraction of implementation of Scenario 1, the failures of communications are generated and analyzed as follows:

$failures(client \parallel server) = failures(e \parallel p)$

$\{\{(\langle\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB, requestC, infoC,$
$resultC\}\},$
$\{(\langle requestA\rangle, X) \mid X \subseteq \{requestA, resultA, requestB, infoB, resultB, requestC, infoC, resultC\}\},$
$\{(\langle requestB\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, resultB, requestC, infoC,$
$resultC\}\},$
$\{(\langle requestC\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB, requestC,$
$resultC\}\},$
$\{(\langle requestA, infoA\rangle, X) \mid X \subseteq \{requestA, infoA, requestB, infoB, resultB, requestC, infoC, resultC\}\},$
$\{(\langle requestB, infoB\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, requestC,$
$infoC, resultC\}\},$
$\{(\langle requestC, infoC\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, itresultB, requestC$
$, infoC\}\},$
$\{(\langle requestA, infoA, resultA\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB, requestC, infoC, resultC\}\},$
$\{(\langle requestB, infoB, resultB\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB,$
$requestC, infoC, resultC\}\},$
$\{(\langle requestC, infoC, resultC\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB,$
$requestC, infoC, resultC\}\},$
$\ldots\}$

In this scenario, three services, *serviceA*,*serviceB* and *serviceC*, can be requested by *client* and offered by *server*. For each type of service, the communications between processes follow the sequence of events *request*, *info* and *service*.

Step 3.b.2: Generate and Analyze Failures of Scenario 2

In this implementation scenario, process *client* has only one port *e*, and process *server* has only one port *p*. Thus, *client* can be represented as {*alphabet*(*e*), *failures*(*e*)}, and *server* can be represented as {*alphabet*(*p*), *failures*(*p*)}.

As we are interested in communications between processes, in the abstraction of implementation of Scenario 2, the failures of communications are generated and analyzed as follows:

$$failures(client \parallel server) = failures(e \parallel p)$$

$$\{\{(\langle\rangle, X) \mid X \subseteq \{requestA, infoA, resultA$$

$$\{(\langle requestA\rangle, X) \mid X \subseteq \{requestA, resultA\}\}, \{(\langle requestA, infoA\rangle, X) \mid X \subseteq \{requestA, infoA\}\},$$

$$\{(\langle requestA, infoA, resultA\rangle, X) \mid X \subseteq \{infoA, resultA\}\},$$

$$\ldots\}$$

In this scenario, only one service, *serviceA*, can be requested by *client* and offered by *server*. The communications between processes follow the sequence of events *requestA*, *infoA* and *serviceA*.

Step 3.b.3: Generate and Analyze Failures of Scenario 3

In this implementation scenario, process *client* has only one port *e*, and process *server* has only one port *p*. Thus, *client* can be represented as {*alphabet*(*e*), *failures*(*e*)}, and *server* can be represented as {*alphabet*(*p*), *failures*(*p*)}.

As we are interested in communications between processes, in the abstraction of implementation of Scenario 3, the failures of communications are generated and analyzed as follows:

$$failures(client \parallel server) = failures(e \parallel p)$$

$$\{\{(\langle\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\}\},$$

$$\{(\langle requestA\rangle, X) \mid X \subseteq \{requestA, resultA, requestB, infoB, resultB\}\},$$

$$\{(\langle requestB\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, resultB\}\},$$

$\{(\langle requestA, infoA\rangle, X) \mid X \subseteq \{requestA, infoA, requestB, infoB, resultB\}\}$,

$\{(\langle requestB, infoB\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB\}\}$,

$\{(\langle requestA, infoA, resultA\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\}\}$,

$\{(\langle requestB, infoB, resultB\rangle, X) \mid X \subseteq \{requestA, infoA, resultA, requestB, infoB, resultB\}\}$,

$\ldots\}$

In this scenario, two services, *serviceA* and *serviceB*, can be requested by *client* and offered by *server*. For each type of service, the communications between processes follow the sequence of events *request*, *info* and *service*.

Illustration of Step 4: Build Categorical Models of Failures from Design

The aim of this step is to construct categories for modeling progress of communications in the design. The progress of communications can be indicated by failures. In Chapter 3, the categories of traces in proposition 3 is provided as follows.

- **Category of Failures**: Each object is of the form *failures* to indicate a process. A Morphism $failures_a \rightarrow failures_b$ means the process with the failures from trace $\langle\rangle$ to the trace a evolves to the process with the failures from trace $\langle\rangle$ to the trace b, where $failures_a \subseteq failures_b$.

Proof of constructing category of failures is provided in Chapter 3.

Proposition 6. DFailures1 is a category. It captures the designed behaviors of the system based on failures extracted from the design in section 5.4.1. In **DFailures**, each object represents failures of communications in the system designed; each morphism models the subset relationship between failures denoted by $\subseteq$ to indicate the progress of the communications; and each identity represents the subset relationship to itself.

Fig. 5.6 illustrates the **DFailures1** category.

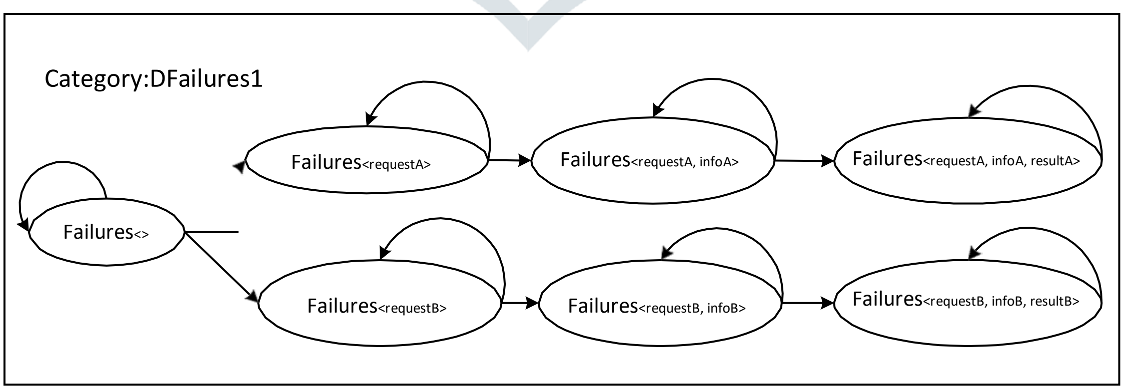

Figure 5.6: Category of Failures from the Design

Proof.

Objects: Each object is the failures of $client \parallel server$ in design. $failures_{\langle event1 \ldots event2 \rangle}$ represents all the failures from trace $\langle\rangle$ to trace $\langle event1 \ldots event2 \rangle$. For example, $failures_{\langle\rangle} = \{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}$ is an object, $failures_{\langle requestA\rangle} =$

$\{\{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}, \{(\langle requestA\rangle, X) \mid$ $\langle requestA\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA\rangle)\}\}$ is an object, and $failures_{\langle requestA, infoA\rangle}$ $= \{\{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}, \{(\langle requestA\rangle, X) \mid \langle requestA\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA\rangle)\}, \{(\langle requestA, infoA\rangle, X) \mid \langle requestA, infoA\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA, infoA\rangle)\}\}$ is an object as well.

Morphisms: Let $failures_x$ and $failure_y$ be objects. If $failures_x \subseteq failures_y$, there is a morphism $failures_x \rightarrow failures_y$. For example, $failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ is a morphism.

Identities: For each object, $failures_m$, there is an identity $failures_m \subseteq failures_m$, which in- dicates $failures_m$ is a subset of itself. For example, $failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA\rangle}$ is an identity.

Composition: Given any morphisms $morph_{x,y} : failures_x \subseteq failures_y$ and $morph_{y,z} : failures_y \subseteq failures_z$, with codomain of $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_x \subseteq failures_y \subseteq failures_z$. Thus, there is a composition morphism: $morph_{y,z} \circ morph_{x,y} : failures_x \subseteq failures_z$. For example, $failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA,infoA\rangle} \circ failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ is a mor- phism, which is $failures_{\langle\rangle} \subseteq failures_{\langle requestA,infoA\rangle}$

Associativity: For all morphisms $morph_{w,x} : failures_w \subseteq failures_x$, $morph_{x,y} : failures_x \subseteq failures_y$ and $morph_{y,z} : failures_y \subseteq failures_z$, with codomain of $morph_{w,x}$ = domain of $morph_{x,y}$ and codomain $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_w \subseteq failures_x \subseteq failures_y \subseteq failures_z$ to represent the subset relationships between failures. Thus, there are $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}) = morph_{y,z} \circ (failures_w \subseteq failures_y) = failures_w \subseteq failures_z$, and$(morph_{y,z} \circ morph_{x,y})$ $\circ\ morph_{w,x} = (failures_x \subseteq failures_z) \circ morph_{w,x} = failures_w \subseteq failures_z$. So, $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}) = (morph_{y,z} \circ morph_{x,y}) \circ morph_{w,x}$. For example, there is $(failures_{\langle requestA,infoA\rangle} \rightarrow failures_{\langle requestA,infoA,resultA\rangle} \circ failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA,infoA\rangle}) \circ failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle} = failures_{\langle requestA,infoA\rangle} \rightarrow failures_{\langle requestA,infoA,resultA\rangle} \circ (failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA,infoA\rangle} \circ failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle})$.

□

Illustration of Step 5: Build Categorical Models of Failures from Abstraction of Implementation

The aim of this step is to construct categories for communications in the abstraction of im- plementation. The progress of communications can be indicated by failures. In Chapter 3, the categories of traces in proposition 3 is provided as follows.

- **Category of Failures**: Each object is of the form *failures* to indicate a process. A Morphism $failures_a \rightarrow failures_b$ means the process with the failures from trace $\langle\rangle$ to the trace a evolves to the process with the failures from trace $\langle\rangle$ to the trace b, where $failures_a \subseteq failures_b$.

Proof of constructing category of failures is provided in Chapter 3.

Step 5.1: Build Categorical Models of Failures from Abstraction of Implementation of Sce- nario 1

Proposition 7. IFailures1 is a category. It captures the behaviors of the system based on failures of communications extracted from the abstraction of implementation of scenario 1 in section 5.4.3. In **IFailures1**, each object represents the failures of communications; each morphism models the subset relationship between failures denoted by $\subseteq$ to indicate the progress of communications; and each identity represents the subset relationship to itself.

Fig. 5.7 illustrates the **IFailures1** category.

Proof.

Objects: Each object is failures of $client \parallel server$ in scenario 1. $failures_{\langle event1 \ldots event2 \rangle}$ repre- sents all the failures from trace $\langle\rangle$ to trace $\langle event1 \ldots event2 \rangle$. For example, $failures_{\langle\rangle} = \{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}$ is an object, $failures_{\langle requestC \rangle} = \{\{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}, \{(\langle requestC \rangle, X) \mid \langle requestC \rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestC \rangle)\}\}$ is an ob- ject, $failures_{\langle requestC, infoC \rangle} = \{\{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}, \{(\langle requestC \rangle, X) \mid \langle requestC \rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel$

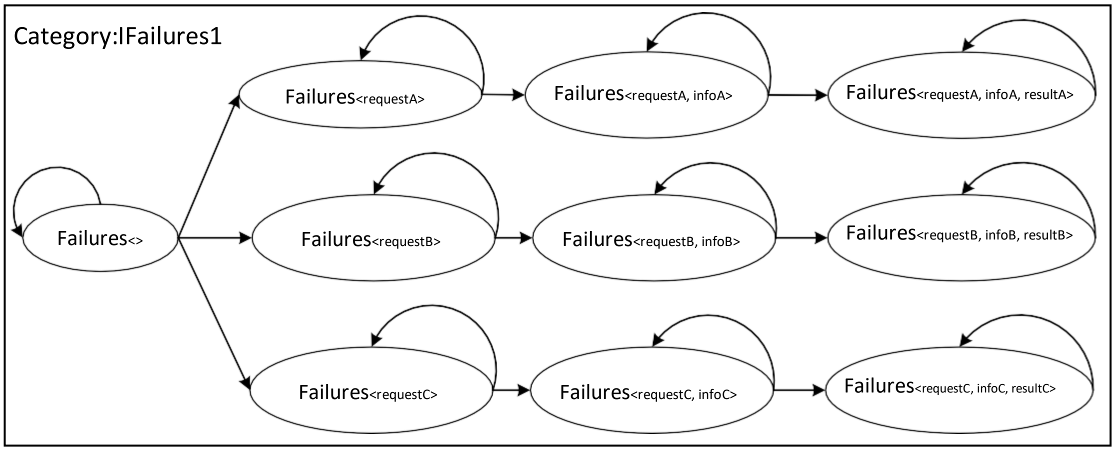

Figure 5.7: Category of Failures from the Abstraction of Implementation of Scenario 1

$server/\langle requestC\ \rangle)\}, \{(\langle requestC\ , infoC\ \rangle, X\ ) \mid \langle requestC\ , infoC\ \rangle \in traces(client\ \parallel server\ ) \wedge\ X\ \in$ $refusals(client\ \parallel server/\langle requestC\ , infoC\ \rangle)\}\}$ is an object as well.

Morphisms: Let $failures_x$ and $failure_y$ be objects. If $failures_x \subseteq failures_y$, there is a morphism $failures_x \rightarrow\ failures_y$. For example, $failures_{\langle\rangle} \rightarrow\ failures_{\langle requestC\ \rangle}$ is a morphism.

Identities: For each object, $failures_m$, there is an identity $failures_m \subseteq failures_m$, which in- dicates $failures_m$ is a subset of itself. For example, $failures_{\langle requestC\ \rangle} \rightarrow failures_{\langle requestC\ \rangle}$ is an identity morphism.

Composition: Given any morphisms $morph_{x,y} : failures_x \subseteq failures_y$ and $morph_{y,z} : failures_y \subseteq failures_z$, with codomain of $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_x \subseteq failures_y \subseteq failures_z$. Thus, there is a composition morphism: $morph_{y,z} \circ morph_{x,y} : failures_x \subseteq failures_z$. For example, $failures_{\langle requestC\ \rangle} \rightarrow\ failures_{\langle requestC,infoC\ \rangle} \circ\ failures_{\langle\rangle} \rightarrow\ failures_{\langle requestC\ \rangle}$ is a mor- phism, which is $failures_{\langle\rangle} \subseteq failures_{\langle requestC,infoC\ \rangle}$

Associativity: For all morphisms $morph_{w,x} : failures_w \subseteq failures_x$, $morph_{x,y} : failures_x \subseteq failures_y$ and $morph_{y,z} : failures_y \subseteq failures_z$, with codomain of $morph_{w,x}$ = domain of $morph_{x,y}$ and codomain $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_w \subseteq failures_x \subseteq failures_y \subseteq failures_z$ to represent the subset relationships between failures. Thus, there are $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}\ ) = morph_{y,z} \circ (failures_w \subseteq failures_y\ ) = failures_w \subseteq failures_z$, and$(morph_{y,z} \circ$ $morph_{x,y}\ ) \circ morph_{w,x} = (failures_x \subseteq failures_z\ ) \circ morph_{w,x} = failures_w \subseteq failures_z$. So, $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}\ ) = (morph_{y,z} \circ morph_{x,y}\ ) \circ morph_{w,x}$. For example, there is

$(failures_{\langle requestC,infoC\ \rangle} \rightarrow failures_{\langle requestC,infoC,resultC\ \rangle} \circ failures_{\langle requestC\ \rangle} \rightarrow failures_{\langle requestC,infoC\ \rangle})$

$$\circ\, failures_{\langle\rangle} \rightarrow failures_{\langle requestC \rangle} = failures_{\langle requestC,infoC \rangle} \rightarrow failures_{\langle requestC,infoC,resultC \rangle} \circ \left(failures_{\langle requestC \rangle} \rightarrow failures_{\langle requestC,infoC \rangle} \circ failures_{\langle\rangle} \rightarrow failures_{\langle requestC \rangle} \right).$$

$\square$

Step 5.2: Build Categorical Models of Failures from Abstraction of Implementation of Scenario 2

Proposition 8. IFailures2 is a category. It captures the behaviors of the system based on failures of communications extracted from the abstraction of implementation of scenario 2 in section 5.4.3. In **IFailures2**, each object represents the failures of communications; each morphism models the subset relationship between failures denoted by $\subseteq$ to indicate the progress of communications; and each identity represents the subset relationship to itself.

Fig. 5.8 illustrates the **IFailures2** category.

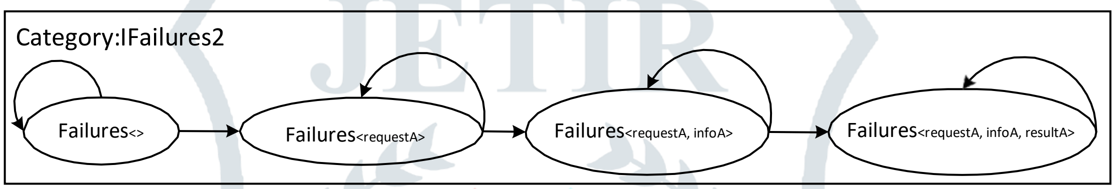

Figure 5.8: Category of Failures from the Abstraction of Implementation of Scenario 2

Proof.

Objects: Each object is failures of $client \parallel server$ in scenario 2. $failures_{\langle event1 \ldots event2 \rangle}$ represents all the failures from trace $\langle\rangle$ to trace $\langle event1 \ldots event2 \rangle$. For example, $failures_{\langle\rangle} = \{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}$ is an object, $failures_{\langle requestA \rangle} = \{\{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}, \{(\langle requestA \rangle, X) \mid \langle requestA \rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA \rangle)\}\}$ is an object, and $failures_{\langle requestA, infoA \rangle} = \{\{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}, \{(\langle requestA \rangle, X) \mid \langle requestA \rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA \rangle)\}, \{(\langle requestA, infoA \rangle, X) \mid \langle requestA, infoA \rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA, infoA \rangle)\}\}$ is an object as well.

Morphisms: Let $failures_x$ and $failure_y$ be objects. If $failures_x \subseteq failures_y$, there is a morphism $failures_x \rightarrow failures_y$. For example, $failures_{\langle\rangle} \rightarrow failures_{\langle requestA \rangle}$ is a morphism.

Identities: For each object, $failures_m$, there is an identity $failures_m \subseteq failures_m$, which indi- cates $failures_m$ is a subset of itself. For example, $failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA\rangle}$ is an identity morphism.

Composition: Given any morphisms $morph_{x,y} : failures_x \subseteq failures_y$ and $morph_{y,z} : failures_y \subseteq failures_z$, with codomain of $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_x \subseteq failures_y \subseteq failures_z$. Thus, there is a composition morphism: $morph_{y,z} \circ morph_{x,y} : failures_x \subseteq failures_z$. For example, $failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA,infoA\rangle} \circ failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ is a mor- phism, which is $failures_{\langle\rangle} \subseteq failures_{\langle requestA,infoA\rangle}$

Associativity: For all morphisms $morph_{w,x} : failures_w \subseteq failures_x$, $morph_{x,y} : failures_x \subseteq failures_y$ and $morph_{y,z} : failures_y \subseteq failures_z$, with codomain of $morph_{w,x}$ = domain of $morph_{x,y}$ and codomain $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_w \subseteq failures_x \subseteq failures_y \subseteq failures_z$ to represent the subset relationships between failures. Thus, there are $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}) = morph_{y,z} \circ (failures_w \subseteq failures_y) = failures_w \subseteq failures_z$, and$(morph_{y,z} \circ morph_{x,y})$ $\circ\ morph_{w,x} = (failures_x \subseteq failures_z) \circ morph_{w,x} = failures_w \subseteq failures_z$. So, $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}) = (morph_{y,z} \circ morph_{x,y}) \circ morph_{w,x}$. For example, there is $(failures_{\langle requestA,infoA\rangle} \rightarrow failures_{\langle requestA,infoA,resultA\rangle} \circ failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA,infoA\rangle}) \circ failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle} = failures_{\langle requestA,infoA\rangle} \rightarrow failures_{\langle requestA,infoA,resultA\rangle} \circ (failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA,infoA\rangle} \circ failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle})$.

□

Step 5.3: Build Categorical Models of Failures from Abstraction of Implementation of Sce- nario 3

Proposition 9. IFailures3 is a category. It captures the behaviors of the system based on failures of communications extracted from the abstraction of implementation of scenario 3 in section 5.4.3. In **IFailures3**, each object represents the failures of communications; each morphism models the subset relationship between failures denoted by $\subseteq$ to indicate the progress of communications; and each identity represents the subset relationship to itself.

Fig. 5.9 illustrates the **IFailures3** category.

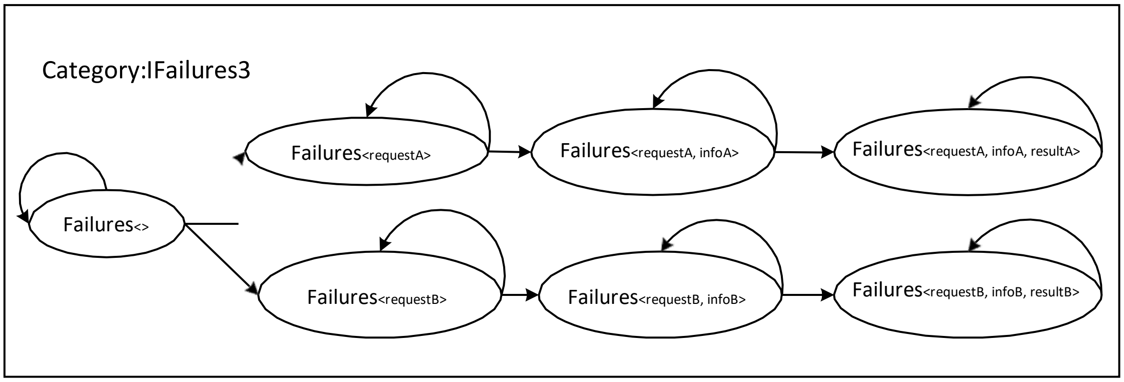

Figure 5.9: Category of Failures from the Abstraction of Implementation of Scenario 3

Proof.

Objects: Each object is failures of $client \parallel server$ in scenario 2. $failures_{\langle event1 \ldots event2 \rangle}$ represents all the failures from trace $\langle\rangle$ to trace $\langle event1 \ldots event2 \rangle$. For example, $failures_{\langle\rangle} = \{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}$ is an object, $failures_{\langle requestA\rangle} = \{\{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}, \{(\langle requestA\rangle, X) \mid \langle requestA\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA\rangle)\}\}$ is an object, and $failures_{\langle requestA,infoA\rangle} = \{\{(\langle\rangle, X) \mid \langle\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle\rangle)\}, \{(\langle requestA\rangle, X) \mid \langle requestA\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA\rangle)\}, \{(\langle requestA, infoA\rangle, X) \mid \langle requestA, infoA\rangle \in traces(client \parallel server) \wedge X \in refusals(client \parallel server/\langle requestA, infoA\rangle)\}\}$ is an object as well.

Morphisms: Let $failures_x$ and $failure_y$ be objects. If $failures_x \subseteq failures_y$, there is a morphism $failures_x \rightarrow failures_y$. For example, $failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ is a morphism.

Identities: For each object, $failures_m$, there is an identity $failures_m \subseteq failures_m$, which indi- cates $failures_m$ is a subset of itself. For example, $failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA\rangle}$ is an identity morphism.

Composition: Given any morphisms $morph_{x,y} : failures_x \subseteq failures_y$ and $morph_{y,z} : failures_y \subseteq failures_z$, with codomain of $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_x \subseteq failures_y \subseteq failures_z$. Thus, there is a composition morphism: $morph_{y,z} \circ morph_{x,y} : failures_x \subseteq failures_z$. For example, $failures_{\langle requestA\rangle} \rightarrow failures_{\langle requestA,infoA\rangle} \circ failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ is a morphism, which is $failures_{\langle\rangle} \subseteq failures_{\langle requestA,infoA\rangle}$

Associativity: For all morphisms $morph_{w,x} : failures_w \subseteq failures_x$, $morph_{x,y} : failures_x \subseteq failures_y$ and $morph_{y,z}$:

$failures_y \subseteq failures_z$, with codomain of $morph_{w,x}$ = domain of $morph_{x,y}$ and codomain $morph_{x,y}$ = domain of $morph_{y,z}$, there is $failures_w \subseteq failures_x \subseteq failures_y \subseteq failures_z$ to represent the subset relationships between failures. Thus, there are $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}) = morph_{y,z} \circ (failures_w \subseteq failures_y) = failures_w \subseteq failures_z$, and$(morph_{y,z} \circ morph_{x,y}) \circ morph_{w,x} = (failures_x \subseteq failures_z) \circ morph_{w,x} = failures_w \subseteq failures_z$. So, $morph_{y,z} \circ (morph_{x,y} \circ morph_{w,x}) = (morph_{y,z} \circ morph_{x,y}) \circ morph_{w,x}$. For example, there is

$$(failures_{\langle requestA,infoA\rangle} \to failures_{\langle requestA,infoA,resultA\rangle} \circ failures_{\langle requestA\rangle} \to failures_{\langle requestA,infoA\rangle}) \circ failures_{\langle\rangle} \to failures_{\langle requestA\rangle} = failures_{\langle requestA,infoA\rangle} \to failures_{\langle requestA,infoA,resultA\rangle} \circ (failures_{\langle requestA\rangle} \to failures_{\langle requestA,infoA\rangle} \circ failures_{\langle\rangle} \to failures_{\langle requestA\rangle}).$$

□

Illustration of Step 6: Construct Functors from Categories of Design to Cate- gories of Abstraction of Implementation

The aim of this step is to verify consistency between design and implementation by construct- ing categories and functors. According to Chapter 3, consistency of communications with failures between the design and the implementation is defined as follows:

Consistency of Communications with Failures: Given a sequence of communications with failures in the design to represent the progress of communications, *DFailures* : $failures_{\langle\rangle} \to$

$failures_{\langle devent1 \rangle} \rightarrow \cdots \rightarrow failures_{\langle devent1, ..., deventn \rangle}$, and a sequence of communications with fail- ures in the implementation to represent the progress of communications, $IFailures : failures_{\langle\rangle} \rightarrow failures_{\langle ievent1 \rangle} \rightarrow \cdots \rightarrow failures_{\langle ievent1, ..., ieventn \rangle}$. If there exists a mapping from *DFailures* to *IFailures* with structure preserved between failures, which can map each trace of $failures_{\langle devent1, ..., devent_i \rangle}$ to the same trace of $failures_{\langle ievent1, ..., ievent_i \rangle}$ with the refusals of the trace of $failures_{\langle devent1, ..., devent_i \rangle}$ being a subset of the refusals of the corresponding trace of $failures_{\langle ievent1, ..., ievent_i \rangle}$, and can map $failures_{\langle devent_1, ..., devent_i \rangle} \rightarrow failures_{\langle devent_1, ..., devent_{i+1} \rangle}$ to $failures_{\langle ievent_1, ..., ievent_i \rangle} \rightarrow failures_{\langle ievent1, ..., ievent_{i+1} \rangle}$, then *IFailures* is consistent with *DFailures*. If all sequences in the design have corresponding mapping sequences in the implementation, the communications in the implementation are consistent with the communications in the design.

To verify consistency of communications with failures between design and implementation, the construction of a functor can be used [55, 56, 57, 58]. If there exists a functor that maps the category of failures from design to the category of failures from implementation, the implementation is con- sistent with the design. Otherwise, the implementation is inconsistent with the design. According to Chapter 3, the functor can be constructed with the following approach.

- For each object, *ocd*, in design, there must be a corresponding object, *oci*, in implementation, such that *ocd* can be mapped to *oci* when each trace in *ocd* has the same trace in *oci*, and the corresponding refusals in *ocd* are a subset of the corresponding refusals in *oci*.
- For each morphism $md : ocd1 \rightarrow ocd2$ in design, there must be a corresponding morphism $mi : oci1 \rightarrow oci2$ in implementation, such that *md* can be mapped to *mi* when *ocd1* and *ocd2* can be mapped to *oci1* and *oci2* respectively.

Step 6.1: Construct Functors from Categories of Design to Categories of Abstraction of Im- plementation of Scenario 1

Based on the analysis of categories **DFailures1** and **IFailures1**, the consistency between the design and the implementation is verified by constructing a functor **DfToIf1**: **DFailures1** → **IFail- ures1**. This functor maps objects and morphisms of **DFailures1** to the corresponding objects and morphisms of **IFailures1** as follows.

- Objects Mapping: let *ocd* be an object of **DFailures1**, and let *oca* be an object of **IFailures1**. As *ocd* and *oca* represent communications with failures, each element in failures is a pair with the form (*trace, refusals*). When each element $\{(t_d, E_d)|t_d$ is a trace $\wedge$ E_d is refusals$\}$ in *ocd* has a corresponding element $\{(t_a, E_a)|t_a$ is a trace $\wedge$ E_a is refusals$\}$ with $t_d = t_a$ and $E_d \subseteq E_a$, there exists a mapping from *ocd* to *oca*. This indicates that all the com- munications between *client* and *server* in design are captured in implementation. For ex- ample, $failures_{\langle requestA\rangle}$ in **DFailures1** in the design represents communications with fail- ures of $\langle\rangle$ and failures of $\langle requestA\rangle$, and there exists $failures_{\langle requestA\rangle}$ in **IFailures1** in the implementation as well. Thus, there is a mapping from $failures_{\langle requestA\rangle}$ in **DFailures1** to $failures_{\langle requestA\rangle}$ in **IFailures1**.

- Morphisms Mapping: For every morphism $mcd : ocd_1 \rightarrow ocd_2$ of **DFailures1**, there must exist one corresponding morphism $mca : oca_1 \rightarrow oca_2$ of **IFailures1**, such that there exists a mapping from *mcd* to *mca* when ocd_1 and ocd_2 can be mapped to oca_1 and oca_2 respectively. These mappings indicate that all the progresses of communications in design are captured in implementation. For example, there exist a mapping from $failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ in **DFailures1** to $failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ in **IFailures1**

- Identities Mapping: By following the objects mapping and morphisms mapping, identity mapping is preserved from **DFailures1** to **IFailures1**.

- Composition Morphisms Mapping: By following the objects mapping and morphisms map- ping, compositions of morphisms mapping are preserved from **DFailures1** to **IFailures1**.

Fig. 5.10 shows that **DfToIf1**: **DFailures1** → **IFailures1** is a functor.

The successful construction of the functor **DfToIf1** indicates that the communications between *client* and *server* in the implementation of scenario 1 and the communications between *client* and *server* in the design are consistent. Though scenario 1 implemented one more service, *serviceC*, which is not specified in the design, all services, *serviceA* and *serviceB* in the design are captured in the implementation.

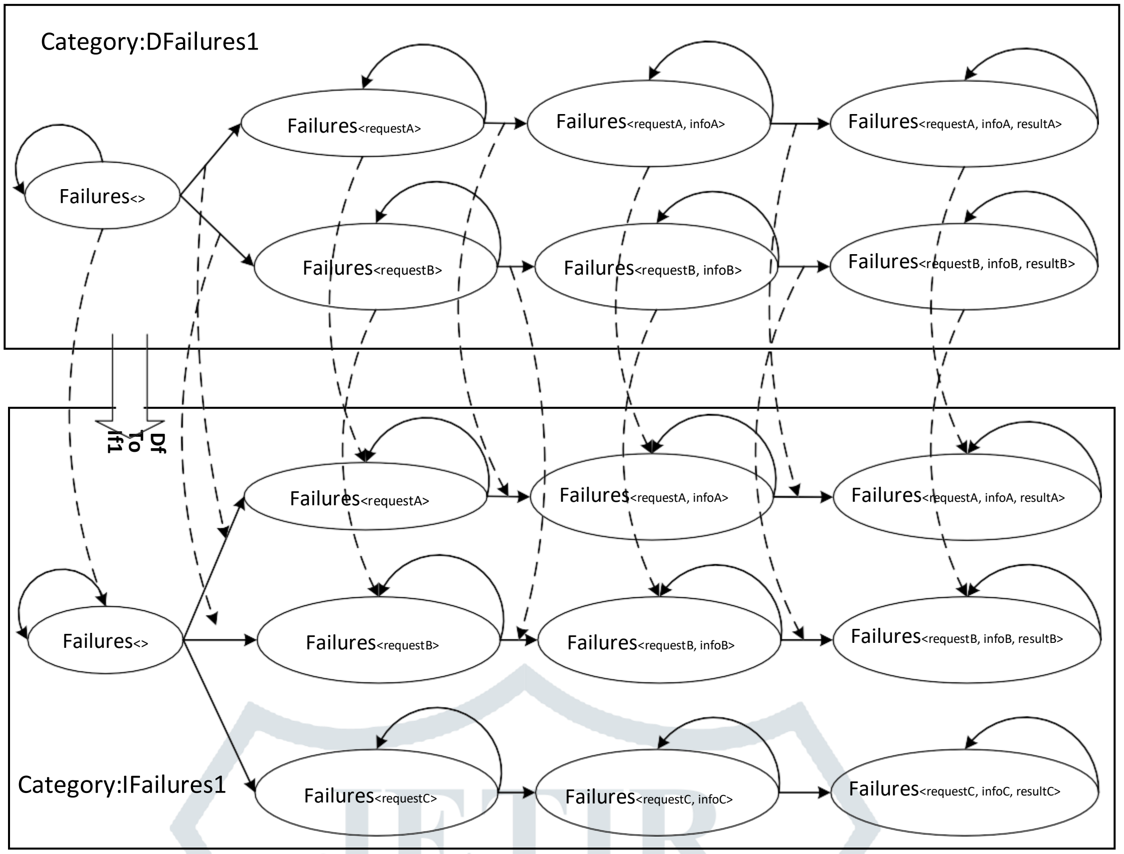

Figure 5.10: A Functor from the Category of Failures in Design to the Category of Failures in the Abstraction of Implementation of Scenario 1

Step 6.2: Construct Functors from Categories of Design to Categories of Abstraction of Implementation of Scenario 2

The implementation of scenario 2 just provides *serviceA*. There is no *serviceB* in the implemen- tation. According to the definition of consistency of communications with failures and the approach of constructing functors, for the categories **DFailures1** and **IFailures2**, we cannot construct a func- tor from **DFailures1** to **IFailures2**. All the objects related to the *serviceB* in **DFailures1** cannot be mapped to any object in **IFailures2**.

Fig. 5.11 shows that we can not construct such a functor from **DFailures1** to **IFailures2**.

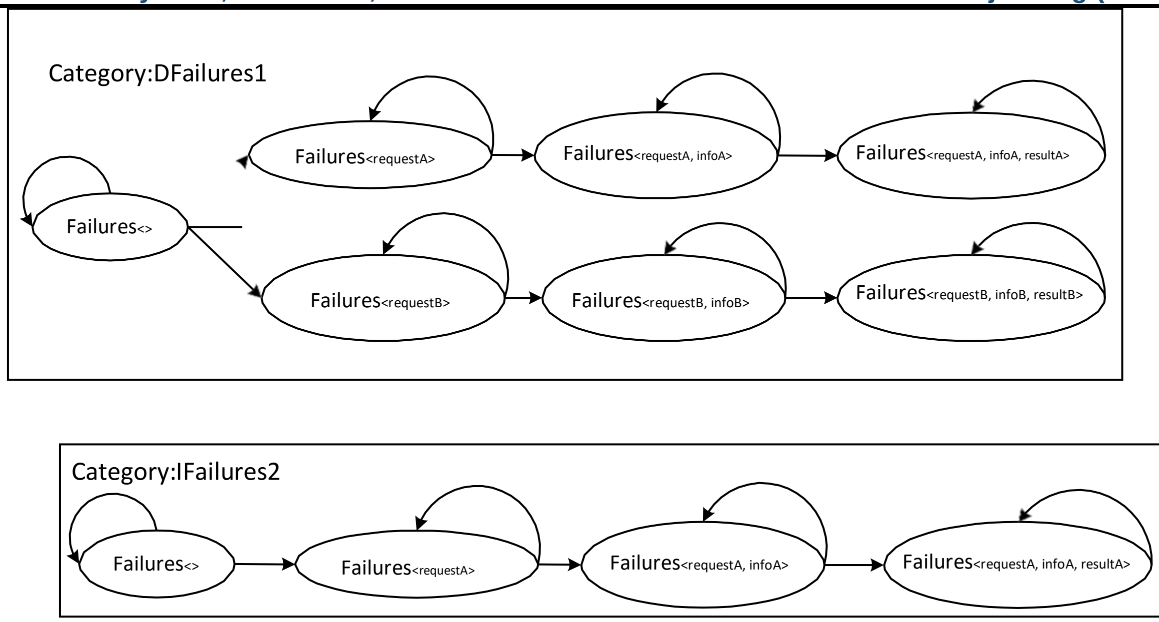

Figure 5.11: No Functor from the Category of Failures in Design to the Category of Failures in the Abstraction of Implementation of Scenario 2

Failing to construct the functor indicates that not all the communications in the design are cap- tured in the implementation. For this scenario, communications related to *serviceB* is not imple- mented. Namely, the communications between *client* and *server* in the design are inconsistent with the communications between *client* and *server* in the implementation.

Step 6.3: Construct Functors from Categories of Design to Categories of Abstraction of Im- plementation of Scenario 3

Scenario 3 implemented all the services in the design. Based on the analysis of categories **DFailures1** and **IFailures3**, the consistency between the design and the implementation is verified by constructing a functor **DfToIf3**: **DFailures1** → **IFailures3**. This functor maps objects and morphisms of **DFailures1** to the corresponding objects and morphisms of **IFailures3** as follows.

- Objects Mapping: let *ocd* be an object of **DFailures1**, and let *oca* be an object of **IFailures3**. As *ocd* and *oca* represent communications with failures, each element in failures is a pair with the form (*trace, refusals*). When each element $\{(t_d, E_d)|t_d$ is a trace $\wedge$ E_d is refusals$\}$

  in *ocd* has a corresponding element $\{(t_a, E_a)|t_a$ is a trace $\wedge$ E_a is refusals$\}$ with $t_d = t_a$ and $E_d \subseteq E_a$, there exists a mapping from *ocd* to *oca*. This indicates that all the com- munications between *client* and *server* in design are captured in implementation. For ex- ample, $failures_{\langle requestA\rangle}$ in **DFailures1** in the design represents communications with fail- ures of $\langle\rangle$ and failures of $\langle requestA\rangle$, and there exists $failures_{\langle requestA\rangle}$ in **IFailures3** in the

implementation as well. Thus, there is a mapping from $failures_{\langle requestA\rangle}$ in **DFailures1** to $failures_{\langle requestA\rangle}$ in **IFailures3**.

- Morphisms Mapping: For every morphism $mcd : ocd_1 \rightarrow ocd_2$ of **DFailures1**, there must exist one corresponding morphism $mca : oca_1 \rightarrow oca_2$ of **IFailures3**, such that there exists a mapping from mcd to mca when ocd_1 and ocd_2 can be mapped to oca_1 and oca_2 respectively. These mappings indicate that all the progresses of communications in design are captured in implementation. For example, there exist a mapping from $failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ in **DFailures1** to $failures_{\langle\rangle} \rightarrow failures_{\langle requestA\rangle}$ in **IFailures3**

- Identities Mapping: By following the objects mapping and morphisms mapping, identity mapping is preserved from **DFailures1** to **IFailures3**.

- Composition Morphisms Mapping: By following the objects mapping and morphisms map- ping, compositions of morphisms mapping are preserved from **DFailures1** to **IFailures3**.

Fig. 5.12 shows that **DfToIf3**: **DFailures1** $\rightarrow$ **IFailures3** is a functor.

The successful construction of the functor **DfToIf3** indicates that the communications between *client* and *server* in the implementation of scenario 3 and the communications between *client* and *server* in the design are consistent.

Summary

In this chapter, the categorical framework is used to verify consistency of communications with failures between design and implementation. This framework used failures, category theory and abstraction of implementation, and is illustrated by a running example with a design and three dif- ferent scenarios of implementation. In doing so, the design of processes communications is modeled

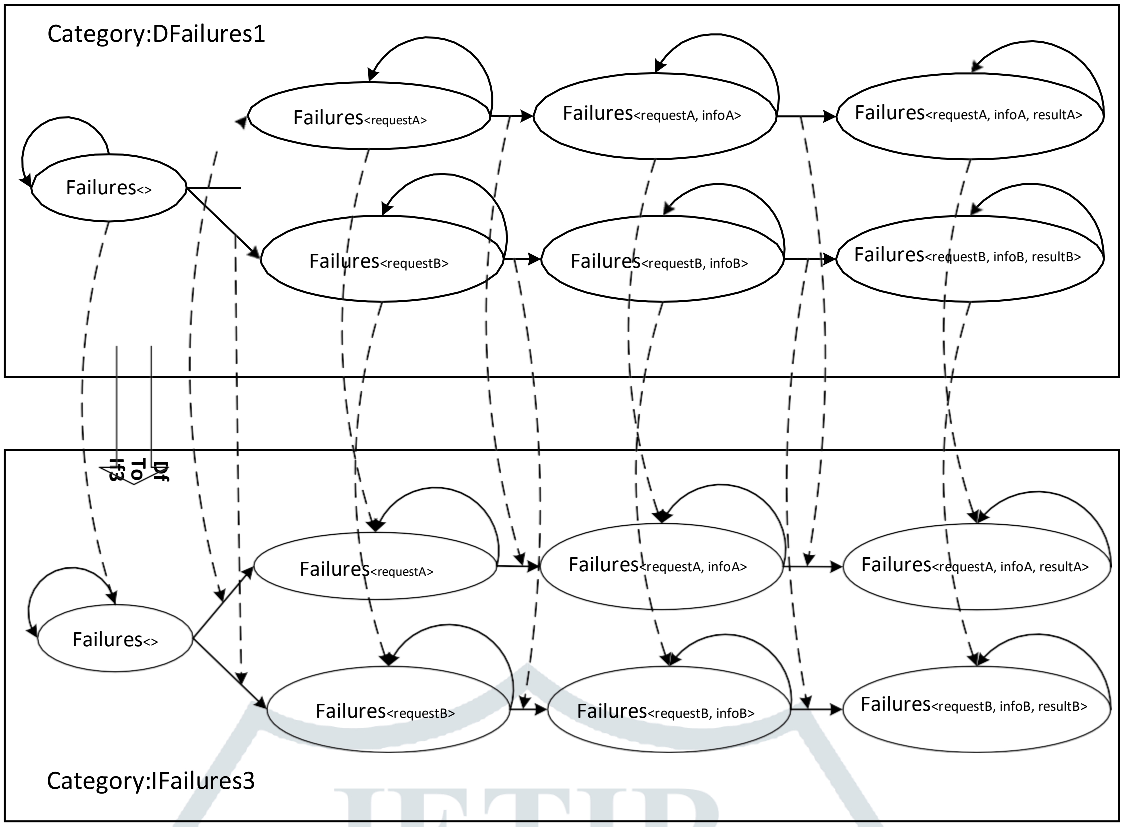

Figure 5.12: A Functor from the Category of Failures in Design to the Category of Failures in the Abstraction of Implementation of Scenario 3

and analyzed by CSP, three scenarios of implementation are created by Erasmus, communications with failures of three scenarios are analyzed based on abstraction, categories of communications with failures from the design and from three scenarios of implementation are created, and, by con- structing functors, the consistency of communications between the design and three scenarios of implementation is verified.

In the next chapter, we introduce algorithms for the categorical framework to verify consistency of communications between design and implementation.

Chapter 6

Algorithms for Verification with Failures

Introduction

To automate the verification of communications, several algorithms are developed for the cate- gorical framework in this chapter. As failures of a process consist of traces, algorithms developed for verification with failures can be used for verification with traces as well. Section 6.2 briefs the contributions in developing algorithms. Section 6.3 gives an overview of algorithms developed for the categorical framework. Section 6.4 introduces data structure and basic functions used for developing algorithms. Section 6.5 provides algorithms for generating failures from abstraction of implementation in Erasmus. Section 6.6 and section 6.7 presents algorithms for constructing categories and functors respectively. Section 6.8 summarizes this chapter.

Contributions

Several contributions in developing algorithms are introduced as follows:

- Basic data structures and functions are developed for algorithms used for verification.
- Algorithms are developed for operations in Erasmus, such as sequential execution, recursion, nondeterministic choice, deterministic choice, and parallel execution.
- Algorithms are developed for constructing categories from failures.

- Algorithms are developed for constructing functors between categories.

The Framework with Algorithms

In Chapter 3, 4, and 5, we proposed the categorical framework and used it to model and analyze the consistency of communications with failures. For CSP, the tool named FDR is developed for generating failures of processes and communications [5]. For Erasmus, we proposed several rules to analyze and generate failures in Chapter 3. Also, we proposed several definitions and approaches to build categories and functors based on failures of processes and communications. In this section, algorithms are developed for Erasmus and categories used in the framework (See Fig. 6.1).

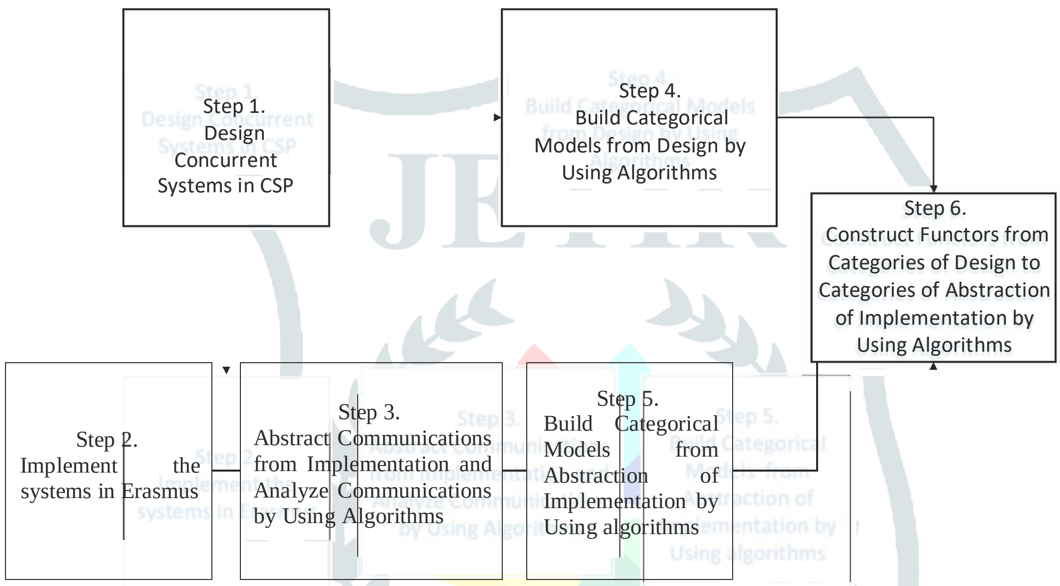

Figure 6.1: The Categorical Framework with Algorithms

(1). In step 3, algorithms are developed for automatically generating failures of process com- munications from abstraction of implementation. These algorithms are used to achieve research objective OBJ3.

(2). In step 4 and step 5, algorithms are developed for automatically generating categories from failures of process communications in design or abstraction of implementation. These algorithms are used to achieve research objective OBJ4 and objective OBJ5.

(3). In step 6, algorithms are developed for automatically constructing functors from categories of design to categories of abstraction of implementation. These algorithms are used to achieve research objective OBJ6.

In the following sections, the algorithms for the above mentioned steps are illustrated in detail.

Data Structures and Basic Functions Used in Algorithms

In this section, we introduce data structures and basic functions used in algorithms for the framework.

Data Structures

As we analyze failures and categories, several notions related to failures and categories are defined with the following data structures.

- An *Event* is represented by a *String*.
- An *EventSet* is a set of events. It is represented by a *Set* of *String*.
- An *Alphabet* is a set of all events of a process. It is represented by a *Set* of *String*.
- A *Trace* is a sequence of events. It is represented by a *List* of *String*.
- A *Refusal* of a trace is a set that contains sets of events. It is represented by a *Set* of *EventSet*.
- A *Failure* is a pair (*Trace*, *Refusal*) that contains a trace and a refusal of the trace. It is represented by a pair with the data structure of *Trace* and the data structure of *Refusal*.
- The *Failures* is a set, and each element of the set is a failure. It is represented by a *Set* of *Failure*.
- A *Process* is a pair (*Alphabet*, *Failures*) that contains an *Alphabet* and *Failures* to represent a process. It is represented by a pair with the data structure of *Alphabet* and the data structure of *Failures*.

- An *Object* is a pair (*Data*, *Children*) to represent a process. It consists of two parts: (1). *Data* contains the information of a process. It is represented by *Failures* of *Process*, (2). *Children* consists of a *List* of *Object*s which morphisms from the *Object* are connected to.

- A *Category* is a category of failures. Each object in the category describes failures of a process. Each morphism between objects indicates an evolution from one process to another. *Object* may have other *Object*s as its *Children*. Always, there is an object *Root* to describe failures of the process with the empty trace.

Basic Functions

In this research, as we analyze failures, several functions related to failures are defined as fol- lows.

- *Boolean evtBelongsEvtSet* (*Event*, *EventSet*) is a function. It takes two inputs, *Event* and *EventSet*, and then returns true if *Event* is in *EventSet*. Otherwise, it returns false.

- *Boolean evtSetBelongsRefusal* (*EventSet*, *Refusal*) is a function. It takes two inputs, *EventSet* and *Refusal*, and then returns true if *EventSet* is in *Refusal*. Otherwise, it returns false.

- *Boolean compareSet* (*Set*, *Set*) is a function. It takes two inputs, *Set* and *Set*, and then returns true if two *Set* are same. Otherwise, it returns false.

- *Boolean compareTrace* (*Trace*, *Trace*) is a function. It takes two inputs, *Trace* and *Trace*, and then returns true if two *Trace* are the same. Otherwise, it returns false.

- *Boolean subSet* (*Set*, *Set*) is a function. It takes two inputs, *Set* and *Set*, and then returns true if the first *Set* is a subset of the second *Set*. Otherwise, it returns false.

- *Boolean subTrace* (*Trace*, *Trace*) is a function. It takes two inputs, *Trace* and *Trace*, and then returns true if the first *Trace* is a prefix of the second *Trace*. Otherwise, it returns false.
- *Trace addPrefixTrace* (*Trace*, *Trace*) is a function. It takes two inputs, *Trace* and *Trace*, and then returns a new *Trace* with the first *Trace* followed by the second *Trace*.
- *Set powerSet* (*Set*) is a function. It takes an input, *Set*, and then returns the power set of the

  Set.

- *Set evtSetFromRefusal* (*Refusal*) is a function. It takes an input, *Refusal*, and then returns a *Set* of *Event*s that contains all *Event*s occurred in *Refusal*.

- *Set setUnion* (*Set*,*Set*) is a function. It takes two inputs, *Set* and *Set*, and then returns a *Set* that is the set union of the first *Set* and the second *Set*.
- *Set setIntersection* (*Set*,*Set*) is a function. It takes two inputs, *Set* and *Set*, and then returns a *Set* that is the set intersection of the first *Set* and the second *Set*.
- *Set setDifference* (*Set*,*Set*) is a function. It takes two inputs, *Set* and *Set*, and then returns a *Set* whose elements are in the first *Set* and the second *Set*, but not in the intersection of the first *Set* and the second *Set*.
- *Set* of *Traces findSuccessfulTraces* (*Process*) is a function. It takes a process, and then returns a *Set* of *Traces* whose elements are successful traces in the process.

Algorithms for Generating Failures from Abstraction of Imple- mentation

In step 3 of the framework, failures are used to model and analyze processes and communica- tions. Several rules on failures are defined to describe the relationships between processes in Chap- ter 3. These rules include sequential execution *failures*($C1$; $C2$), recursion *failures*(**loop**$\{C\}$), non- deterministic choice *failures*(**case**$\{C1|. . . |Cn\}$), deterministic choice *failures*(**select**$\{C1|. . . |Cn\}$), and parallel execution *failures*($C1 \parallel C2$). In this section, we propose algorithms for the abovemen- tioned rules as follows.

Sequential Execution

Given two Erasmus statements $C1$ and $C2$, a sequence $C1$; $C2$ means the process behaves as $C1$ first, then behaves as $C2$ after $C1$ executed successfully. In chapter 3, the rule for calculating failures of $C1$; $C2$ is proposed as follows.

$$\begin{aligned} failures(C1; C2) = &\{(s, X) \mid (s, X) \in failures(C1)\} \\ &\cup \{(s^\frown t, X) \mid s^\frown\langle C\rangle \in traces(C1) \wedge (t, X) \in failures(C2)\} \end{aligned}$$

Based on this rule, we propose the Algorithm 1 for sequential execution ; as follows. In Algo- rithm 1, as $C1$ and $C2$ may have different alphabets, the alphabet of $C1$; $C2$ is the set union of the alphabet of $C1$ and the alphabet of $C2$. For $C1$, the refusal of each failure needs to be updated, as $C1$ may refuse to execute some events from $C2$. For $C2$, the refusal of each failure needs to be updated, as $C2$ may refuse to execute some events from $C1$. After updating, the refusals of all failures with successful traces in $C1$ are replaced by using the refusal of the failure with $\langle\rangle$ trace in $C2$, then $C1$ is added into $C1$; $C2$. Subsequently, traces of all failures in $C2$ are updated by adding each successful trace in *C*1 as their trace prefix. In the last, *C*2 is added into *C*1; *C*2 with removing the failure with $\langle\rangle$ trace in *C*2. The function *findSuccessfulTraces* is used to find all successful tracesin *C*1.

Algorithm 1 *sequentialExecution*
Input: Process *C*1, Process *C*2

Output: Process *R*

1: create an empty process *R*

2: the alphabet of *R* ← *setUnion* (the alphabet of *C*1, the alphabet of *C*2)

3: extend the refusal of each failure in failures of *C*1 by using the alphabet of *C*2 4: extend the refusal of each failure in failures of *C*2 by using the alphabet of *C*1 5: create a set of traces *sucC1TraceSet* ← *findSuccessfulTraces* (*C*1)

6: create a failure *initC2Failure* ← the failure with $\langle\rangle$ trace in *Failures* of *C*2

7: **for** each trace *sucC1Trace* in *sucC1TraceSet* **do** 8: **for** each failure *c1Failure* in *Failures* of *C*1 **do** 9: **if** *Trace* of *c1Failure* = *sucC1Trace* **then**

10: *Refusal* of *c1Failure* ← *Refusal* of *initC2Failure*

11: **end if**

12: **end for**

13: **end for**

14: *Failures* of *R* ← *Failures* of *C*1

15: remove *initC2Failure* from *Failures* of *C*2

16: **for** each trace *sucC1Trace* in *sucC1TraceSet* **do**

17: **for** each failure *c2Failure* in *Failures* of *C*2 **do**

18: trace *c2Trace* of *c2Failure* ← *addPrefixTrace* (*sucC1Trace*,trace *c2Trace* of *c2Failure*)

19: **end for**

20: **end for**

21: *Failures* of *R* ← (*Failures* of *R*) ∪ (*Failures* of *C*2)

22: **return** *R*

In lines 7, 8, 16 and 17 of the Algorithm 1, there are **for** loops. In lines 7 and 8, each failure with the trace in *sucC1TraceSet* of process *C*1 will be modified. In lines 16 and 17, each trace in process *C*2 will be modified by adding each trace in *sucC1TraceSet* of process *C*1. Thus, the complexity of the Algorithm 1 would be $O(n^2)$, where *n* is the number of traces or failures in a process.

Recursion

Given an Erasmus statement $C1$, a recursion **loop**$\{C1\}$ means the process $C1$ executes re- peatedly. Namely, once a $C1$ finishes execution successfully, another $C1$ will start execution. In Chapter 3, the rule for calculating failures of **loop**$\{C1\}$ is proposed as follows.

$$
\begin{aligned}
failures(\mathbf{loop}\{C\}) =\{(s, X) \mid (s, X) \in failures(C)\} \\
\cup\ \{(s^{1}{}^\frown s, X) \mid s^{1}{}^\frown\langle \mathrm{C}\rangle \in traces(C) \wedge\ (s, X) \in failures(C)\} \\
\cup\ \ldots \cup\ \{(s^{1}{}^\frown s^{2}{}^\frown \ldots {}^\frown s^{n-1}{}^\frown s^{n}, X) | s^{i}{}^\frown\langle \mathrm{C}\rangle \in traces(C) \\
\wedge\, 1 \le i \le n-1 \wedge (s, X) \in (failures(C))\}
\end{aligned}
$$

Based on this rule, we propose Algorithm 2 for recursion **loop**$\{C1\}$ as follows. In this algo- rithm, $C1$ repeats a specific number of times, and we use the sequential execution Algorithm 1 to calculate the recursion.

Algorithm 2 *recursion*
Input: Process $C1$, Integer *repeatTime*

Output: Process R

```
1: create an empty process R
2: Integer i ← 0
3: if repeatTime ≥ 1 then
4:     for i from 1 to repeatTime do
5:         R ← sequentialExecution (R, C1)
6:     end for
7: end if
8: return R
```

In line 4 of the Algorithm 2, there is a **for** loop that will call the Algorithm 1 a specific number of times. As the complexity of Algorithm 1 $O(n^2)$, the complexity of the Algorithm 2 is $O(n^3)$.

Nondeterministic Choice

In Erasmus, the nondeterministic choice **case** means the choice of actions is made internally by the process and is not determined by the environment. In Chapter 3, given two processes $C1$ and $C2$, the rule for calculating failures of **case**$\{C1 \mid C2\}$ is proposed as follows.

$failures(\textbf{case}\{C1|C2\}) = \{(s, X)|(s, X) \in failures(C1 \cup \; failures(C2)\}$

Based on this rule, we propose Algorithm 3 for nondeterministic choice **case**$\{C1 \mid C2\}$ as fol- lows. In the Algorithm 3, as $C1$ and $C2$ may have different alphabets, the alphabet of **case**$\{C1|C2\}$ is the set union of the alphabet of $C1$ and the alphabet of $C2$. For $C1$, the refusal of each failure needs to be updated, as $C1$ may refuse to execute some events from $C2$. For $C2$, the refusal of each failure needs to be updated, as $C2$ may refuse to execute some events from $C1$. After this, the algorithm sets the *failures* of **case**$\{C1 \mid C2\}$ to be the *failures* of $C1$, then add *failures* of $C2$ into *failures* of **case**$\{C1 \mid C2\}$.

Algorithm 3 *nondeterministicChoice*

Input: Process $C1$, Process $C2$

Output: Process R

1: create an empty process R

2: the alphabet of $R \leftarrow setUnion$ (the alphabet of $C1$, the alphabet of $C2$)

3: extend the refusal of each failure in failures of $C1$ by using the alphabet of $C2$ 4: extend the refusal of each failure in failures of $C2$ by using the alphabet of $C1$ 5: failures of $R \leftarrow$ (failures of $C1$) $\cup$ (failures of $C2$)

6: **return** R

In the Algorithm 3, failures of **case**$\{C1 \mid C2\}$ is calculated by the union of failures of $C1$ and failures of $C2$. Thus, the complexity of the Algorithm 3 is $O(n)$, where n is the number of failures in a process.

Deterministic Choice

In Erasmus, the deterministic choice **select** means the choice of actions is made externally by the environment and is not determined by the process itself. In Chapter 3, given two processes $C1$ and $C2$, the rule for calculating failures of $failures(\textbf{select}\{C1|C2\})$ is defined as follows.

$$failures(\textbf{select}\{C1|C2\}) = \{(s, X)|(s = \langle\rangle \wedge \; (s, X) \in failures(C1) \cap \; failures(C2))$$
$$\vee (s \mathrel{/}= \langle\rangle \wedge (s, X) \in failures(C1) \cup failures(C2))\}$$

Based on this rule, we propose Algorithm 4 for nondeterministic choice **select** for deterministic choice as follows.

In the Algorithm 4, as $C1$ and $C2$ may have different alphabets, the alphabet of **select**$\{C1 \mid C2\}$ is the set union of the alphabet of $C1$ and the alphabet of $C2$. For $C1$, the refusal of each failure needs to be updated, as $C1$ may refuse to execute some events from $C2$. For $C2$, the refusal of each failure need sto be updated, as $C2$ may refuse to execute some events from $C1$. After this, the algorithm calculates the intersection of the refusal of failure with $\langle\rangle$ trace in $C1$ and the refusal of failure with $\langle\rangle$ trace in $C2$, adds a failure with $\langle\rangle$ trace and the refusal intersection into **select**$\{C1 \mid C2\}$, and then add $C1$ without the failure containing $\langle\rangle$ trace and $C2$ without the failure containing $\langle\rangle$ trace into **select**$\{C1 \mid C2\}$.

Algorithm 4 *deterministicChoice*

Input: Process $C1$, Process $C2$

Output: Process R

create an empty process R

the alphabet of $R \leftarrow$ *setUnion* (the alphabet of $C1$, the alphabet of $C2$)

extend the refusal of each failure in failures of $C1$ by using the alphabet of $C2$

extend the refusal of each failure in failures of $C2$ by using the alphabet of $C1$

create a refusal *c1Refusal* $\leftarrow$ the refusal of failure with $\langle\rangle$ trace in $C1$

create a refusal *c2Refusal* $\leftarrow$ the refusal of failure with $\langle\rangle$ trace in $C2$

create a refusal *c1qRefusal* $\leftarrow$ *setIntersection* (*c1Refusal* , *c2Refusal*)

create a failure *c1c2Failure* $\leftarrow$ ($\langle\rangle$, *c1c2Refusal*)

$C1 \leftarrow$ remove the failure with $\langle\rangle$ trace in $C1$

$C2 \leftarrow$ remove the failure with $\langle\rangle$ trace in $C2$

failures of $R \leftarrow$ (failures of R) + *c1c2Failure*

failures of $R \leftarrow$ (failures of R) $\cup$ (failures of $C1$)

failures of $R \leftarrow$ (failures of R) $\cup$ ($C2$)

return R

In the Algorithm 4, **select**$\{C1|C2\}$ is calculated by modifying the failure with empty trace and by using the union of $C1$ and $C2$. Thus, The complexity of the Algorithm 4 is $O(n)$, where n is the number of failures in a process.

Parallel Execution

Given two processes $C1$ and $C2$, parallel execution $\parallel$ describes two processes communicatewith each other. Both process must agree on all actions that occur. In Chapter 3, the rule for calculating failures of $C1 \parallel C2$ is proposed as follows.

$$failures(C1 \parallel C2) = \{(s, X \cup Y) | ((s, X) \in failures(C1) \wedge (s, Y) \in failures(C2))\}$$

Based on this rule, we propose Algorithm 5, Algorithm 6 and Algorithm 7 for parallel execution $\parallel$ as follows. In the algorithm *parallelExecution*, as $C1$ and $C2$ has different alphabets, the alphabet of $C1 \parallel C2$ is the set union of the alphabet of $C1$ and the alphabet of $C2$. For $C1$, the refusalof each failure needs to be updated, as $C1$ may refuse to execute some events from $C2$. For $C2$, the refusal of each failure needs to be updated, as $C2$ may refuse to execute some events from $C1$. After this, the Algorithm 6 calculates the failure with trace $\langle\rangle$ of $C1 \parallel C2$, and then it uses the Algorithm 7 to calculate the failures of next actions of $C1 \parallel C2$.

Algorithm 5 *parallelExecution*
Input: Process $C1$, Process $C2$

Output: Process R

create an empty process R

the alphabet of $R \leftarrow$ *setUnion* (the alphabet of $C1$, the alphabet of $C2$)

extend the refusal of each failure in failures of $C1$ by using the alphabet of $C2$

extend the refusal of each failure in failures of $C2$ by using the alphabet of $C1$

failures of $R \leftarrow$ *buildInitCommunication*($C1$, $C2$)

failures of $R \leftarrow$ (failures of R) $\cup$ *buildNextCommunication*($\langle\rangle$,$C1$,$C2$);

return R

Algorithm 6 *buildInitCommunication*
Input: Process $C1$, Process $C2$

Output: Set of Failures F

create an empty set of failures F

create a set of failures *c1Failures* $\leftarrow$ *Failures* of $C1$

create a set of failures *c2Failures* $\leftarrow$ *Failures* of $C2$

for each failure *c1Failure* in *c1Failures* **do**

  if the trace of *c1Failures* = $\langle\rangle$ **then**

    for each failure *c2Failure* in *c2Failures* **do**

      if the trace of *c2Failure* = $\langle\rangle$ **then**

        create an empty failure *newFailure*

        refusal of *newFailure* $\leftarrow$ *setUnion*(refusal of *c1Failure*, refusal of *c2Failure*)

        trace of *newFailure* $\leftarrow$ $\langle\rangle$

        $F \leftarrow F + newFailure$

      end if

    end for

14: **end if**

15: **end for**

16: **return** *F*

Algorithm 7 *buildNextCommunication*
Input: Trace *t*, Process *C*1, Process *Q*

Output: Set of Failures *F*

1: create an empty set of failures *F*

2: create a set of failures *c1Failures* ← *Failures* of *C*1 3: create a set of

failures *c2Failures* ← *Failures* of *C*2 4: **for** each failure *c1Failure* in

c1Failures **do**

5: **if** *subTrace* (*t*, trace of *c1Failure*) **and** size of *t* + 1 = size of trace of *c1Failure* **then**

6: **for** each failure *c2Failure* in *c2Failures* **do**

7: **if** *subTrace* (*t*, trace of *c2Failure*) **and** size of *t* + 1 = size of trace of *c2Failure* **then**

8: **if** *compareTrace* (trace of *c1Failure*, trace of *c2Failure*) **then**

9: create an empty failure *newFailure*

10: refusal of *newFailure* ← *setUnion*(refusal of *c1Failure*, refusal of *c2Failure*)

11: trace of *newFailure* ← trace of *c2Failure*

12: *F* ← *F* + *newFailure*

13: *F* ← *F* ∪ *buildNextCommunication*(trace of *newFailure*,*C*1,*C*2);

14: **end if**

15: **end if**

16: **end for**

17: **end if**

18: **end for**

19: **return** *F*

In lines 4 and 6 of the Algorithm 6, there are **for** loops to calculate the failure with trace $\langle\rangle$ of $C1 \parallel C2$. The complexity of Algorithm 6 is $O(n^2)$, where n is the number of failures in a process. To calculate the failures of communications after the trace $\langle\rangle$, the Algorithm 7 uses **for** loops in lines 4 and 6, and recursively calls itself in line 13. The complexity of the Algorithm 7 is $O(n^3)$, where n is the number of failures in a process. As the Algorithm 5 uses the Algorithm 6 and the Algorithm 7, the complexity of Algorithm 5 is $O(n^3)$.

Algorithms for Constructing Categories

In step 4 and step 5 of the framework, categories are built from failures of processes generated from design and abstraction of implementation. In Chapter 3, the category of failures is specified in definition 3.8.2 as follows.

Category of Failures: **Category of Failures**: Each object is of the form *failures* to indicate a process. A Morphism $failures_a \rightarrow failures_b$ means the process with the failures from trace $\langle\rangle$ to the trace *a* evolves to the process with the failures from trace $\langle\rangle$ to the trace *b*, where $failures_a \subseteq failures_b$.

Based on this definition, we propose Algorithm 8 and Algorithm 9 to construct categories as follows. In the Algorithm 8, a category can be built for a process to represent the evolving progress of the process. The category is a tree-like structure with root to represent the process with the empty trace. Each morphism between objects indicates an evolution from one process to another. The Algorithm 8 first builds the root, and then uses the Algorithm 9 to build objects after the root.

Algorithm 8 *buildCategoryFromProcess*
Input: Process *P*

Output: Category *R*

```
create an empty category R
for each failure f in failures of P do
    if Trace of f = ⟨⟩ then
        Data of Root of R ← (Data of Root of R) + f
    end if
end for
children of Root of R ← buildChildrenNodes (Root of R, P )
return R
```

Algorithm 9 *buildChildrenNodes*
Input: Object *obj*, Process *p*

Output: List of Objects *chs*

```
create an empty list of object chs
Trace trace ← the Longest Trace in Data of obj
for each failure f in failures of p do
    if trace is the subtrace of the trace t of f and size of trace + 1 = size of t then
        create an empty object child
        Data of child ← Data of obj + f
        children of child ← buildChildrenNodes (child , p)
```

```
8:         chs ← chs + child
9:     end if
10: end for
11: return chs
```

In line 2 of the Algorithm 8, there is a **for** loop to calculate build the root object for the process with empty trace. In lines 3 and 7 of the Algorithm 9, there are a **for** loop and a recursive call to calculate the children of objects that are connected by morphisms. The complexity of the Algorithm 9 is $O(n^2)$, where n means the number of failures in a process or the number of objects in the category. As the Algorithm 8 uses the Algorithm 9, the complexity of the Algorithm 8 is $O(n^2)$.

Algorithms for Constructing Functors

As functor can be used to check structure preserving between two categories, in this research, functors are used to verify consistency of communications with traces and failures between design and implementation. Successful construction of such functor means the process communications in the implementation is consistent with the process communications in the design. Failing to construct such functor could indicate an inconsistency between the design and the implementation.

To construct functors from categories of failures in design to categories of failures in implementation, in Chapter 3, an approach for the construction is introduced as follows.

For each object, *ocd*, in design, there must be a corresponding object, *oci*, in implementation, such that *ocd* can be mapped to *oci* when each trace in *ocd* has the same trace in *oci*, and the corresponding refusals in *ocd* are a subset of the corresponding refusals in *oci*.

- For each morphism $md : ocd1 \rightarrow ocd2$ in design, there must be a corresponding morphism $mi : oci1 \rightarrow oci2$ in implementation, such that *md* can be mapped to *mi* when *ocd1* and *ocd2* can be mapped to *oci1* and *oci2* respectively.

Based on this approach, we propose algorithms for constructing functors as follows. In the Algorithm 10, it uses the Algorithm 11 and the Algorithm 12 to compare root objects and children objects in two categories. In the Algorithm 11, we can compare the trace and refusal of the object in the category of design to the trace and refusal of the object in the category of implementation by following the above mentioned approach for the construction. In the Algorithm 12, each child object in the category of design is compared with corresponding object in the category of implementation.

Algorithm 10 *functor*

Input: Category *dsg* , Category *imp*

Output: Boolean

if *compareTwoObjects*(*Root* of *dsg* , *Root* of *imp*) **then**
    if *compareChildrenObjects*(*Root* of *dsg* , *Root* of *imp*) **then**
        return true
    end if
end if
return false

Algorithm 11 *compareTwoObjects*

Input: Object *dsgObj* , Object *impObj*

Output: Boolean

create failures *dsgP* ← *Data* of *dsgObj*
create failures *impP* ← *Data* of *impObj*
create boolean *flag*
for each failure *dsgF* in *dsgP* **do**
    flag ← false
    for each failure *impF* in *impP* **do**
        if trace of *dsgF* = trace of *impF* **and** refusal of *dsgF* ⊆ refusal of *impF* **then**
            flag ← true
            break
        end if
    end for
    if *flag* =false **then**
        return false
    end if
end for
return true

Algorithm 12 *compareChildrenObjects*

Input: Object *dsgObj* , Object *impObj*

Output: Boolean

create a list of objects *dsgChildren* ← *Children* of *dsgObj*
create a list of objects *impChildren* ← *Children* of *impObj*
create boolean *flag*
for each object *dsgChild* in *dsgChildren* **do**

```
5:    flag ←  false
6:    for each object impChild in impChildren do
7:       if compareTwoObject (dsgChild ,impChild ) then
8:          flag ←  true
9:          if size of children of dsgChild > 0 then
10:            flag ←   compareChildrenObjects(dsgChild , impChild )
11:            break
12:         end if
13:      end if
14:   end for
15:   if flag=false then
16:   return false
17:  end if
18: end for
19: return true
```

In lines 4 and 6 of the Algorithm 11, there are **for** loops used to compare two objects. The complexity of Algorithm 11 is $O(n^2)$, where n is the number of failures in a process. To compare children objects in two categories, the Algorithm 12 uses **for** loops in lines 4 and 6, calls the Algorithm 11 in line 7, and recursively calls itself in line 10. The complexity of the Algorithm 12 is $O(n^4)$, where n is the number of objects in a category. As the Algorithm 10 uses the Algorithm 11 and the Algorithm 12, the complexity of the Algorithm 10 is $O(n^4)$.

Summary

In this chapter, we propose several algorithms for generating failures, categories, and func- tors. In step 3 of the framework, algorithms are developed for automatically generating processes from abstraction of implementation, which include generating failures from sequential execution, recursion, nondeterministic choice, deterministic choice, and parallel execution in the abstraction of implementation in Erasmus. In step 4 and step 5 of the framework, algorithms are developed for generating categories of failures from design and abstraction of implementation. In step 6, algo- rithms are developed for constructing functors from categories of failures in design to categories of failures in abstraction of implementation.

In the next chapter, we introduce verification between communications in implementation and properties of communications in Erasmus. In Erasmus, communications in implementation must conform to properties of communications.

Chapter 7

Verifying Properties of Communications

Introduction

In order for processes to communicate, communications in implementation need to conform to properties of communications in Erasmus. To support our research goal to build the categorical framework for verification, in this chapter, verification between communications in implementation and properties of communications in Erasmus is proposed and introduced. Section 7.2 briefs the contributions in verifying properties of communications. Section 7.3 gives two properties of com- munications that Erasmus implementation must follow. Section 7.4 introduces the methodology for verifying communications in implementation against properties of communications in Erasmus. Section 7.5 provides a running example to illustrate the application of the methodology for verifica- tion. Section 7.6 summarizes this chapter.

Contributions

Several contributions in verifying properties of communications are introduced as follows:

- A methodology is proposed for verifying communications in implementation against proper- ties of communications in Erasmus.

- Data flow analysis is used to abstract and model communications in implementation.
- Category theory is used to model properties of communications in Erasmus and model the abstraction of communications based on data flow analysis.
- Functors are used to verify communications in implementation against properties of commu- nications in Erasmus.

Properties of Communications in Erasmus

Erasmus is a process-oriented programming language, which is based on the idea of CSP but with some differences [18, 21, 22, 25]. An Erasmus program consists of *cells*, *processes*, *ports*, *protocols* and *channels*. A cell, containing a collection of one or more processes or cells, provides the structuring mechanism for an Erasmus program. A process is a self-contained entity which performs computations, and communicates with other processes through its ports. A port, which is of a type of protocol, usually serves as an interface of a process for sending and receiving messages. A protocol specifies the type and the orderings of messages that can be sent and received by ports of the type of this protocol. A channel, which is of a type of protocol, must be built between two ports for two processes to communicate. Erasmus also offers operations for deterministic choices and nondeterministic choices by using keywords *select* and *case* respectively.

In Erasmus, communication is as important as method invocation in object-oriented languages. If two processes p_1 and p_2 want to communicate, they must satisfy some requirements listed in Chapter 2. In this chapter, we focus on the following two properties:

- The *ProcessesCommunication* property: Request are sent by a process through its client port (declared with '–'), then received at channel in of a channel and sent out by channel out of the channel, finally received by the other process at the server port (declared with '+').

- The *Protocols* property: Given a client port π_1 of protocol t_1 and a server port π_2 of protocol t_2, if π_1 and π_2 can communicate, t_2 must satisfy t_1. Here, t_2 satisfies t_1 is defined as that the set of types of requests of t_1 must be a subset of the set of types of requests of t_2, denoted by $t_1 \subseteq t_2$.

- This means that, for any implementation in Erasmus, communications between processes in the implementation must conform to the *ProcessesCommunication* and *Protocols* properties.

Methodology

To ensure implemented communications conform to the properties, we propose a methodology to model and verify communications against properties in Erasmus. The methodology consists of the following steps, each of which is discussed in detail later.

(1) Step 1. Categorize Communications Properties: In this step, we need to model the properties of communications by using category theory.

(2) Step 2. Abstract Communications in Implementation Based on Data Flow Analysis: In this step, we need to use data flow to analyze communications in implementation, and generate abstraction based on data flow analysis.

(3) Step 3. Categorize Abstraction of Communications: In this step, we need to model the ab- straction of communications based on data flow analysis by using category theory.

(4) Step 4. Verify Categories of Communications properties and Categories of abstraction of Communications: In this step, we need to construct functors to verify the categorical models of communications to the categorical models of communications properties.

To illustrate the process of verifying communications against properties, the process steps are demonstrated on a running example.

Illustration of a Example

To illustrate the methodology for verifying communications against properties, a *Hello World* example is developed. In the following code, a message “Hello World” is sent from process *person* via client port $r1$ of protocol $t1$, forwarded through channel c of protocol $t1$, and received by process *world* via server port $r2$ of protocol $t2$. Protocol $t1$ is satisfied by protocol $t2$, denoted by $t1 \subseteq t2$, as request1: Word is a subset of request1: Word | request2 : Word.

line 1: t1 = protocol { request1 : Word }

line 2: t2 = protocol { request1 : Word | request2 : Word } line 3: person = process r1 : - t1 { line 4: r1.request1 = "Hello World";

line 5: }

line 6: world = process r2 : + t2 {

line 7: message : Word = r2.request1; line 8: }

line 9: sample = cell {

line 10: c : t1; person(c); world(c); line 11: }

With the application of the methodology for verification to this example, we are able to verify whether communications in implementation conforms to communications properties.

Illustration of Step 1: Categorize Communications Properties

For a communication to exist between two processes, *ProcessesCommunication* and *Protocols* properties must be satisfied. The aim of this step is to formalize these two properties by using category theory.

Proposition 10. ProcCom is a category to model ProcessCommunication property. Its objects are *process with client port, client port, channel in, channel out, server port*, and *process with server port*; its morphisms between objects represent passing requests from one object to another object; and its identity morphism on each object represents no action on the object.

Proof. (Fig. 7.1, in part, shows that **ProcCom** is a category)

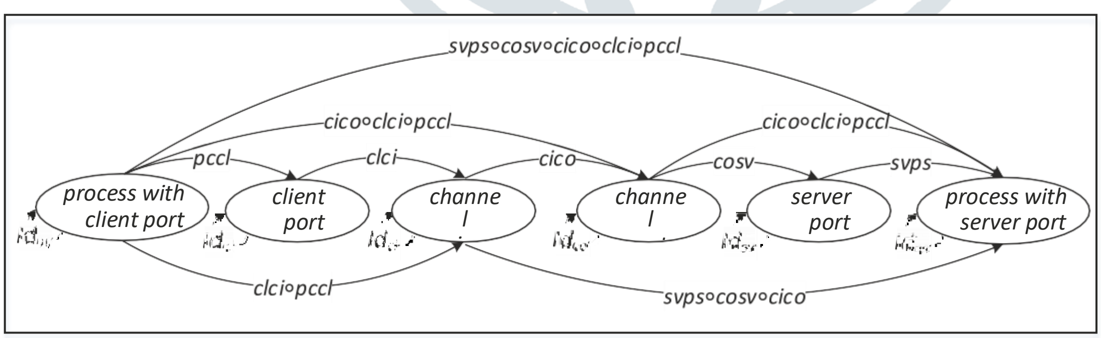

Figure 7.1: The ProcCom category

Objects: *process with client port*, *client port*, *channel in*, *channel out*, *server port*, and *process with server port*.

Morphisms: *pccl* : *process with client port* → *client port*, *clci* : *client port* → *channel in*, *cico* : *channel in* → *channel out* , *cosv* : *channel out* → *server port*, *svps* : *server port* → *process with server port*, each of which represents passing requests from one object to another object.

Identity morphisms: Id_{pc} : *process with client port* → *process with client port* , Id_{cl} : *client port* → *client port*, Id_{ci} : *channel in* → *channel in*, Id_{co} : *channel out* → *channel out* , Id_{sv} : *server port* → *server port*, Id_{ps} : *process with server port* → *process with server port* , each of which represents idle(no action) on the object.

Composition: Given any morphisms $m_1 : obj_a \rightarrow obj_b$ and $m_2 : obj_b \rightarrow obj_c$, with codomain of m_1 = domain of m_2, there is composition morphism: $m_2 \circ m_1 = obj_a \rightarrow obj_c$. In Fig. 7.1, one of the composition morphisms, $svps \circ cosv \circ cico \circ clci \circ pccl$, is shown, which represents requestscan be sent from *process with client port* to *process with server port*.

Associativity: For all morphisms $m_1 : obj_a \rightarrow obj_b$, $m_2 : obj_b \rightarrow obj_c$ and $m_3 : obj_c \rightarrow obj_d$, with codomain of m_1 = domain of m_2 and codomain of m_2 = domain of m_3, there are $m_3 \circ (m_2 \circ m_1) = m_3 \circ (obj_a \rightarrow obj_c) = obj_a \rightarrow obj_d$, and $(m_3 \circ m_2) \circ m_1 = (obj_b \rightarrow obj_d) \circ m_1 = obj_a \rightarrow obj_d$. Thus, $m_3 \circ (m_2 \circ m_1) = (m_3 \circ m_2) \circ m_1$. In Fig. 7.1, one example ofmorphisms with associativity, $(svps \circ cosv \circ cico) \circ (clci \circ pccl) = (svps \circ cosv) \circ (cico \circ clci \circ pccl)$, is shown.

□

Proposition 11. Procls is a Category to model the Protocols property. Its objects are protocols defined in Erasmus program; its morphism represents the ⊆ relation between objects, which is one protocol is satisfied by another protocol; its identity morphism on each object represents the ⊆ relation between the object and itself.

Proof. (Fig. 7.2, in part, shows that **Procls** is a category)

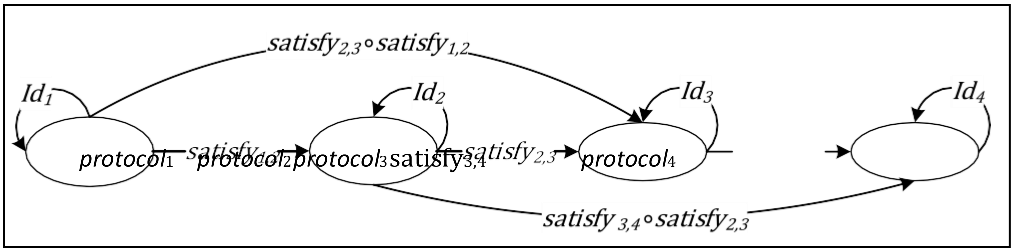

Figure 7.2: A Sample Procls Category

Objects: Each object represents a protocol. Such as, $protocol_1$, $protocol_2$, . . . , $protocol_n$

Morphisms: Let $protocol_x$ and $protocol_y$ be objects. If $protocol_x \subseteq protocol_y$, there is a mor- phism $satisfy_{x,y}$: $protocol_x \to protocol_y$. The morphism represents the subset relation between objects, which is one protocol is satisfied by another protocol (see Protocols property in Section 2.3).

Identities: For each object, $protocol_m$, there is an identity $Id_m : protocol_m \to protocol_m$, which indicates $protocol_m \subseteq protocol_m$. The identity morphism represents the subset relation between object and itself.

Composition: Given any morphisms $satisfy_{x,y} : protocol_x \to protocol_y$ and $satisfy_{y,z} : protocol_y \to protocol_z$, with codomain of $satisfy_{x,y}$ = domain of $satisfy_{y,z}$, there is $protocol_x \subseteq protocol_y \subseteq protocol_z$. Thus, there is a composition morphism: $satisfy_{y,z} \circ satisfy_{x,y} : protocol_x \to protocol_z$. In Fig. 7.2, two of the composition morphisms, $satisfy_{2,3} \circ satisfy_{1,2}$ and $satisfy_{4,5} \circ satisfy_{2,3}$, are shown.

Associativity: For all morphisms $satisfy_{w,x} : protocol_w \to protocol_x$, $satisfy_{x,y} : protocol_x \to protocol_y$ and $satisfy_{y,z} : protocol_y \to protocol_z$, with codomain of $satisfy_{w,x}$ = domain of $satisfy_{x,y}$ and codomain $satisfy_{x,y}$ = domain of $satisfy_{y,z}$, there is $protocol_w \subseteq protocol_x \subseteq protocol_y \subseteq protocol_z$. Thus, there are $satisfy_{y,z} \circ (satisfy_{x,y} \circ satisfy_{w,x}) = satisfy_{y,z} \circ$

$(protocol_w \to protocol_y) = protocol_w \to protocol_z$, and$(satisfy_{y,z} \circ satisfy_{x,y}) \circ satisfy_{w,x} = (protocol_x \to protocol_z) \circ satisfy_{w,x} = protocol_w \to protocol_z$. So, $satisfy_{y,z} \circ (satisfy_{x,y} \circ satisfy_{w,x}) = (satisfy_{y,z} \circ satisfy_{x,y}) \circ satisfy_{w,x}$. In Fig. 7.2, one example of morphisms with as- sociativity, $satisfy_{3,4} \circ (satisfy_{2,3} \circ satisfy_{1,2}) = (satisfy_{3,4} \circ satisfy_{2,3}) \circ satisfy_{1,2}$, is shown. □

Illustration of Step 2: Abstract Communications in Implementation Based onData Flow Analysis

The aim of this step is to abstract communications in implementation based on data flow anal- ysis. Since our interests are in communications, an abstraction is created for extracting the code pertaining only to communications. For the purpose of abstraction, the Definition-Use data flow analysis is employed for tracing requests sent and received by processes via ports and channels. By a data flow analysis, a program of Erasmus can be translated to a data flow graph, where each node represents a statement fragment (that can either be an entire statement or a part of statement) and each edge represents flow of requests between nodes.

The following notations are used for nodes in the data flow graph: (1). **Defining Node of Sending Request** (*DEFR*(*r, p, n* : *f*)) is a node, where the request to be sent is assigned to port *r* in process *p* in the statement fragment *f* in line *n*. (2). **Usage Node of Receiving Request** (*USER*(*r, p, n* : *f*)) is a node, where the request received at port *r* is used in process *p* in the statement fragment *f* in line *n*. (3). **Node of Channel for Receiving Request** (*CRR*(*c, r, p, n* : *f*)) is a node, where the channel *c* connected to port *r* of process *p* is used for receiving incoming request in statement fragment *f* in line *n*. (4). **Node of Channel for Sending Request** (*CSR*(*c, r, p, n* : *f*)) is a node, where the channel *c* connected to port *r* of process *p* is used for sending outgoing request in statement fragment *f* in line *n*.

The data flow graph for the *Hello World* example is represented in Fig. 7.3.

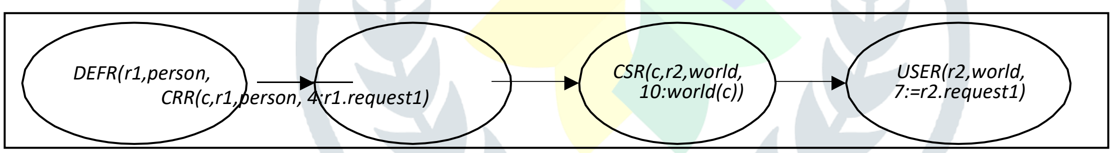

Figure 7.3: Data Flow Graph for The Hello World Example

In this example, firstly data is defined in *r1:request1* in line 4 and assigned to port *r*1 in process *person,* secondly the data is received at channel *c* in line 10, thirdly the data is sent out at channel *c* in line 10, and fourthly the data is received by port *r*2 in process *world* and used in line 7.

Illustration of Step 3: Categorize Abstraction of Communications

In the data flow graph, requests flow along the direction of edge from node *A* to node *B*, with the arrow indicating the direction of flow. This indicates the relation between nodes that the time of the execution of node *A* is earlier than the time of execution of node *B*. The nodes and edges in data flow graph can be formalized using category theory.

Proposition 12. ComNodes is a category for the data flow graph of the *Hello World* example.

Its objects represent the nodes in the dataflow graph; its morphisms represent "execute before or simultaneously", indicated by " ; and its identity morphism on each object represents no action on the object.

Proof. (Fig. 7.4, in part, shows that **ComNodes** is a category)

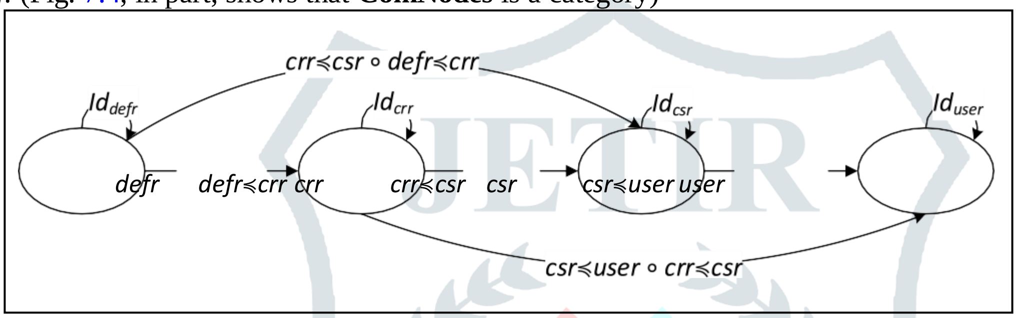

Figure 7.4: The ComNodes category

Objects: *defr* represents node *DEFR*(*r1* , *person, 4* : *r1* : *request1*), *crr* represents node *CRR*(*c*, *r1* , *person, 10* : *person*(*c*)), *csr* represents node *CSR*(*c*, *r2* , *world, 10* : *world* (*c*)), *user* represents node *USER*(*r2* , *world, 7* : *r2* .*request1*).

Morphisms: *defr* " *crr* : *defr* → *crr* , *crr* " *csr* : *crr* → *csr* , *csr* " *usr* : *itcsr* → *user* , each of which represents """ relation between the order of execution of objects.

Identity morphisms: Id_{defr} : *defr* → *defr* , Id_{crr} : *crr* → *crr* , Id_{csr} : *csr* → *csr* , Id_{usr} : *usr* → *usr* , each of which represents the execution of the object is """ to the execution of itself.

Composition: Given any morphisms *x* " *y*: *x* → *y* and *y* " *z* : *y* → *z*, and with codomain of *x* " *y* = domain of *y* " *z*, there is *x* " *y* " *z*. Thus, there is a composition morphism: *y* " *z* ◦ *x* " *y*: *x* → *z*. In Fig. 7.4, two of the composition morphisms, *crr* " *csr* ◦ *defr* " *crr* and *csr* " *user* ◦ *crr* " *csr* , are shown.

Associativity: For all morphisms *w* " *x* : *w* → *x*, *x* " *y* : *x* → *y*, and *y* " *z* : *y* → *z*, with codomain of *w* " *x* = domain of *x* " *y* and codomain *x* " *y* = domain of *y* " *z*, there is *w* " *x* " *y* " *z*. Thus, there are *y* " *z* ◦ (*x* " *y* ◦ *w* " *x*) = *y* " *z* ◦ (*w* → *y*) = *w* → *z*, and (*y* " *z* ◦ *x* " *y*) ◦ *w* " *x* = (*z* → *x*) ◦ *w* " *x* = *w* → *z*. So, *y* " *z* ◦ (*x* " *y* ◦ *w* "*x*) = (*y* " *z* ◦ *x* " *y*) ◦ *w* " *x*. In Fig. 7.4, one example of morphisms with associativity,(*csr* " *user* ◦ *crr* " *csr*) ◦ *defr* " *crr* = *csr* " *user* ◦ (*crr* " *csr* ◦ *defr* " *crr*), is shown.

□

Illustration of Step 4: Verify Categories of Communications Properties and Categories of Abstraction of Communications

The aim of this step is to verify consistency between design and implementation by construct- ing categories and functors. If a property of Erasmus is satisfied by implementation, there must exist a functor that maps the category of the property to the category of abstraction of implemen- tation. Failing to construct such functor could indicate an inconsistency between the implemented system and the specified communication property. The following propositions are used to verify the consistency between the properties and implementation for the *Hello World* Example.

Illustration of Step 4.1: Verify ProcessesCommunication Property

To verify that if all communications conform to the *ProcessesCommunication* property, each time, two processes with their ports and the channel involved in the communication are modeled as a subcategory of the category of data flow graph of the program, then verify if there is a functor from the ProcCom category to the subcategory.

Construct Subcategories

SubPCNodes is a subcategory of **ComNodes**. Its objects are objects from **ComNodes**, which are *defr* , *crr* , *csr* ,*user* ; its morphisms are morphisms from **ComNodes** on those objects, which are
defr “ *crr* , *crr* “ *csr* , and *csr* “ *user* ; and its identities are identities from **ComNodes**, which
are Id_{defr} , Id_{crr} , Id_{csr} , and Id_{user} .

Proof. (Fig. 7.5, in part, shows that **SubPCNodes** is a subcategory)

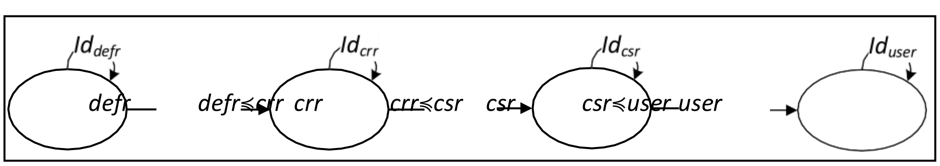

Figure 7.5: The SubPCNodes Category

As **SubPCNodes** contains all the nodes, morphisms, and identities of **ComNodes**, any com- position morphism of **SubPCNodes** also exists in **ComNodes**. Thus, definitely **SubPCNodes** is a

subcategory of **ComNodes**. In Fig. 7.5, composition morphisms are not shown explicitly. □

Since the *Hello World* example has only two processes, only one subcategory is created for the example, which is exactly like the category of the data flow graph of the program. If a program has more processes, a corresponding subcategory should be created for each two of them in the communication.

Construct Functors

FPC: **ProcCom** → **SubPCNodes** is a functor. Fig. 7.6, in part, shows that **FPC** is a functor

This functor can be constructed with the following approach.

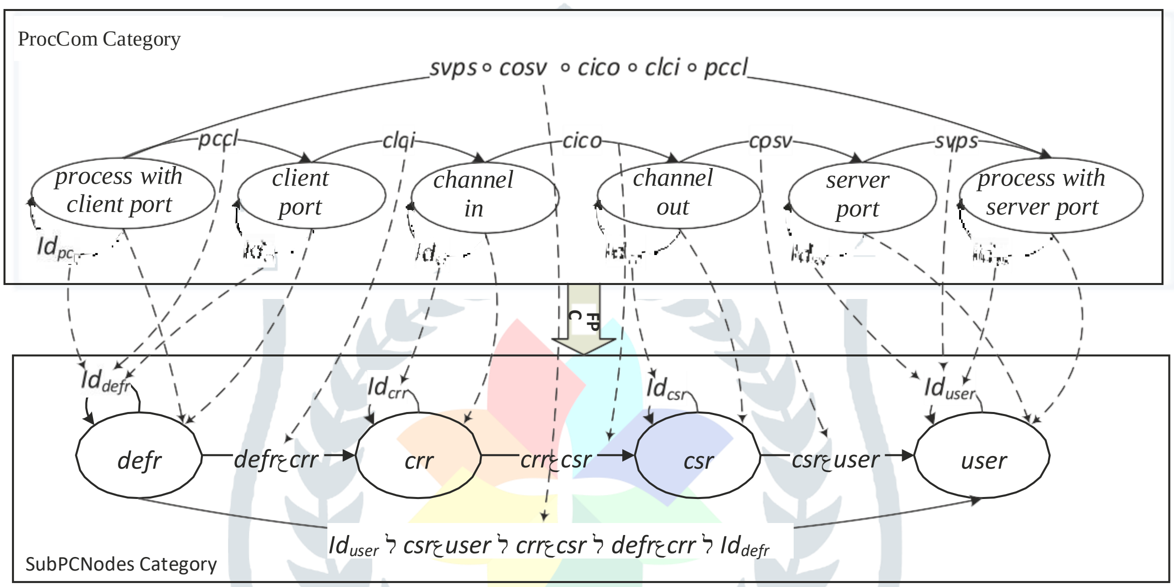

Figure 7.6: The FPC functor

Objects Mapping: (1). As *defr* contains the information of process *person* and client port *r*1, *process with client port* maps to *defr* , and *client port* maps to *defr* . (2). As *crr* contains the information of channel *c* with connection to client port *r*1, *channel in* maps to *crr* . (3). As *csr* contains the information of channel *c* with connection to server port *r*2, *channel out* maps to *csr* . (4). As *user* contains the information of process *world* and server port *r*2, *process with server port user* and *server port* maps to *user* .

Morphisms Mapping: *pccl* maps to Id_{defr} , *clci* maps to *defr* “ *crr* , *cico* maps to *crr* “ *csr* , *cosv* maps to *csr* “ *user* , and *svps* maps to Id_{user} .

Identities Mapping: Id_{pc} maps to Id_{defr} , Id_{cl} maps to Id_{defr} , Id_{ci} maps to Id_{crr} , Id_{co} maps to Id_{csr} , Id_{sv} maps to Id_{user} , and Id_{ps} maps to Id_{user} .

Composition Morphisms Mapping: Given any morphisms $morp_1 : x \to y$ and $morp_2 : y \to z$ of **ProcCom**, wiht codomain of x “ y= domain of y “ z, $morp_1$ maps to x^J “ y^J, $morp_2$ maps to y^J “ z^J, and x maps to x^J, y maps to y^J, z maps to z^J, where x^J “ y^J and y^J “ z^J in **SubPCNodes**, with codomain of x^J “ y^J= domain of y^J “ z^J. As there are a composition morphism:

$morp_2 \circ morp_1 : x \to z$ in **ProcCom**, and a composition morphism: y^J “ $z^J \circ x^J$ “ $y^J : x^J \to z^J$ in **SubPCNodes**, thus $morp_2 \circ morp_1$ maps to y^J “ $z^J \circ x^J$ “ y^J. In Fig. 7.6, one of the composition morphisms mappings, ($svps \circ cosv \circ cico \circ clci \circ pccl$) maps to ($Id_{user} \circ csr$ “ $user \circ crr$ “ $csr \circ defr$ “ $crr \circ Id_{defr}$), is shown.

As functor **FPC** is successfully constructed, the implementation of the *Hello World* example conforms to ProcessesCommunication property.

Illustration of Step 4.2: Verify Protocols Property

To verify that if all communications conform to the Protocols property, each time, the client port and the server port involved in the communication are modeled as a subcategory of the category of data flow graph of the program, then verify if there is a functor from the category of protocols of the program to the subcategory.

Construct Subcategories

According to proposition 2, **ProtlHW** is a category that models protocols used by ports in the *Hello World* Example. Fig. 7.7, in part, shows that **ProtlHW** is a category. Its objects are $t1$ and $t2$, which represent the protocol $t1$ and protocol $t2$; its morphism is $satisfy_{t1\ ,t2} : t1 \to t2$, which represents $t1 \subseteq t2$; its identities are $Id_{t1} : t1 \to t1$ and $Id_{t2} : t2 \to t2$, which represents $t1 \subseteq t1$ and $t2 \subseteq t2$. In Fig. 7.7, composition morphisms are not shown explicitly.

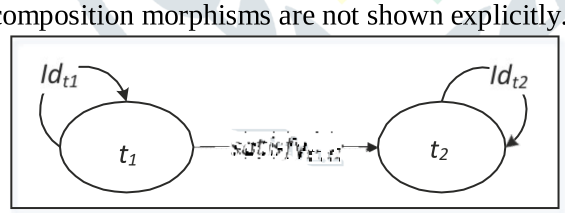

Figure 7.7: The ProtlHW Category

Proposition 13. SubPTNodes is a subcategory of **ComNodes**, which models the client port and the server port involved in the communication.

Proof. (Fig. 7.8, in part, shows that **SubPTNodes** is a subcategory)

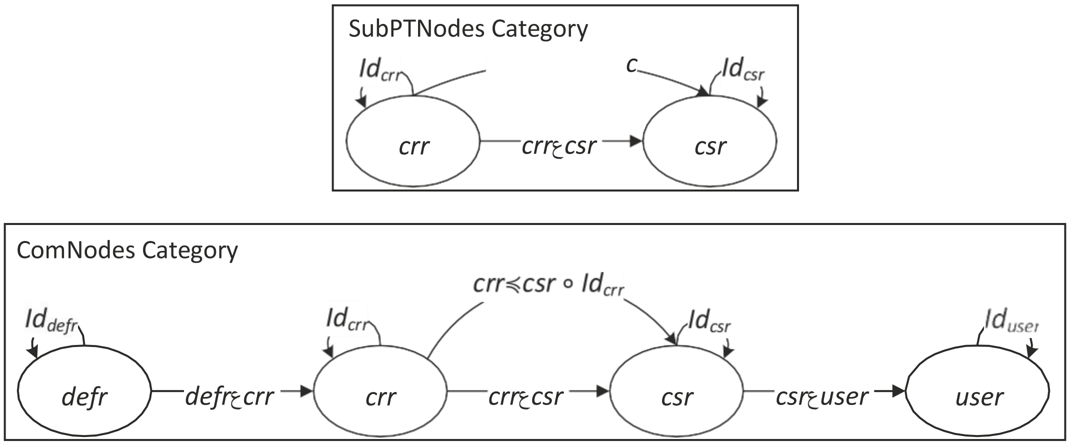

Figure 7.8: The SubPTNodes Category

Objects: *crr* and *csr* of **SubPTNodes** are objects of **ComNodes**, which represents port $r1$ and port $r2$ respectively.

Morphisms: *crr* “ *csr* of **SubPTNodes** is the morphism *crr* “ *csr* of **ComNodes**

Identities: Id_{crr} and Id_{csr} of **SubPTNodes** are identities of **ComNodes**

Composition: Given any morphisms x “ $y : x \to y$ and y “ $z : y \to z$ of **SubPTNodes**, with codomain of x “ y= domain of y “ z, there is x “ y “ z. Thus, there is a composition morphism: y “ $z \circ x$ “ $y : x \to z$ in **SubPTNodes**. Since all objects and morphisms of **SubPTNodes** are objects and morphisms of **ComNodes** respectively, the composition morphism y “ $z \circ x$ “ $y : x \to z$ also exists in **ComNodes**. In Fig. 7.8, one of the composition morphisms of **SubPTNodes**, *crr* “ *csr* $\circ\, Id_{crr}$, is shown. It is from the composition morphism *crr* “ *csr* $\circ\, Id_{crr}$ of **ComNodes**.

□

Since the *Hello World* example has only two ports in the communication, one subcategory is created for the example. If a program has more ports, a corresponding subcategory should be created for each two ports involved in the communication.

Construct Functors

FPT: **ProtlHW** → **SubPTNodes** is a functor. Fig. 7.9, in part, shows that **FPT** is a functor. This functor can be constructed with the following approach.

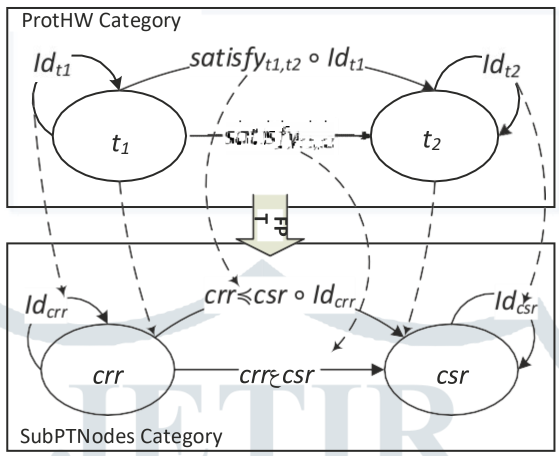

Figure 7.9: The FPT Functor

Objects Mapping: (1). As *crr* contains the information of client port *r*1 of protocol *t*1, *t*1 maps to *crr*. 2) As *csr* contains the information of server port *r*2 of protocol *t*2, *t*2 maps to *csr*.

Morphisms Mapping: $satisfy_{t1\ ,t2}$ maps to *crr* “ *csr* . Identities Mapping: Id_{t1} maps to Id_{crr}, and Id_{t2} maps to Id_{csr}.

Composition Morphisms Mapping: Given any morphisms $morp_1 : x \rightarrow y$ and $morp_2 : y \rightarrow z$ of **ProtlHW**, with codomain of $morp_1$ = domain of $morp_2$. $morp_1$ maps to x^{J} “ y^{J}, $morp_2$ maps to y^{J} “ z^{J}, and x maps to x^{J}, y maps to y^{J}, z maps to z^{J}, where x^{J} “ y^{J} and y^{J} “ z^{J} in **SubPTNodes**, with codomain of x^{J} “ y^{J} = domain of y^{J} “ z^{J}. As there are a composition morphism: $morp_2 \circ morp_1 : x \rightarrow z$ in **ProtlHW**, and a composition morphism: y^{J} “ $z^{J} \circ x^{J}$ “ y^{J} : $x^{J} \rightarrow z^{J}$ in **SubPTNodes**, thus $morp_2 \circ morp_1$ maps to y^{J} “ $z^{J} \circ x^{J}$ “ y^{J}. In Fig. 7.9, one of the composition morphisms mappings, $satisfy_{t1\ ,t2} \circ Id_{t1}$ maps to *crr* “ $csr \circ Id_{crr}$, is shown.

As functor **FPT** is successfully constructed, the implementation of the *Hello World* example conforms to Protocols property.

Summary

This chapter introduces a methodology based on category theory and data flow analysis for mod- eling and verifying properties of communications in Erasmus. To explain the methodology, a simple *Hello World* program implemented in Erasmus is chosen. With the application of this methodolo- gy to the program, its feasibility is successfully proved. In particular, this chapter introduces two properties of communications, abstracts the program with data flow analysis, constructs categories of these properties and abstractions of the program, and verifies consistency between properties and the program with functors.

In the next chapter, we summarize the research contributions by providing conclusion and pro- pose possible future work.

Chapter 8

Conclusion and Future Work

This chapter summarizes the research in this thesis by providing conclusion and possible future work. In Section 8.1, we provide the conclusion from the research in this thesis. Section 8.2 reviews some possible future work.

Conclusion

This research aims to verify the consistency between design and implementation of concurrent systems developed by process-oriented programming languages. To achieve the goal, we proposed an innovative framework to verify consistency of process communications by using CSP, Erasmus, abstract interpretation, data flow analysis, and category theory.

Specifically, several innovative contributions are introduced in this thesis as follows:

- An categorical framework for verification is proposed.
- Rules for abstracting implementation in Erasmus are proposed.
- Rules for analyzing traces and failures from abstraction of implementation in Erasmus are proposed.
- Category theory is used to model communications in design and implementation.
- Functors are used to verify consistency of communications between design and implementa- tion.
- Algorithms are developed for analyzing process operations in Erasmus, such as sequential execution, recursion, nondeterministic choice, deterministic choice, and parallel execution.
- Algorithms are developed for constructing categories from failures of processes.
- Algorithms are developed for constructing functors between categories.
- A methodology is proposed for verifying communications in implementation against proper- ties of communications in Erasmus.
- Data flow analysis is used to abstract and model communications in implementation.
- Category theory is used to model properties of communications in Erasmus and model the abstraction of communications based on data flow analysis.
- Functors are used to verify communications in implementation against properties of commu- nications in Erasmus.

Directions For Future Research

Our work suggests several directions for future work. These directions are as follows.

Using Monoidal Category to Model Communications

Many of the categories have a binary operation on objects and arrows. A monoidal category is a category equipped with a category C, a binary operator bifunctor $\otimes : C \times C \to C$, and a unit u, which satisfies associativity, left identity, right identity and coherence conditions [48]. In a monoidal category, it uses a bifunctor to take two objects in a category and yield an object in the same category. The allowance of this concept to the present work is that, in this field of research, there often are several binary operations on processes, such as sequential execution, deterministic choice, nondeterministic choice, and parallel execution. Each of these binary operations takes two processes and generates a process. For some operations, there may exist a process acting as the unit. For example, the process

STOP is a unit in the deterministic choice operation, such that

P Q *STOP* = *P*. The similarities between monoidal category and binary operations on processes inspire us to work on the direction of using monoidal category to model process communications in future.

Analyzing communications with temporal constraints is a future direction of our research as well. Temporal constraints were proposed by lamport [59], which introduced the “happens before” relation, denoted by “$\rightarrow$”. As its name implies, $e_1 \rightarrow e_2$ if event e_1 happens before, or occurs previously to, event e_2. The “happens before” relation is a strict partial order [59]. When compared with traces in CSP, temporal constraints focus on ”happen before” relation between events, while traces record the possible sequences of events occurred. With temporal constraints, in some cases, it may not be necessary to build completed traces of events, as partial order from temporal constraints could indicate the ordering of events in traces.

Bibliography

[1] P. Welch. Life of occam-Pi. In *Communicating Process Architectures 2013*. Open Channel Publishing Ltd., 2013.

[2] A. T. Sampson, P. Welch, D. Warren, P. Andrews, J. Bjørndalen, S. Stepney, and J. Timmis. Investigating patterns for the process-oriented modelling and simulation of space in complex systems. In *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 17–24, August 2008.

[3] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8): 666–677, 1978.

[4] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Englewood Cliffs, United States, 1985.

[5] A. W. Roscoe. *Understanding concurrent systems*. Springer, London, United Kingdom, 2010.

[6] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, United States, 1999.

[7] A. T. Sampson. *Process-oriented patterns for concurrent software engineering*. PhD thesis, University of Kent, Kent, United Kingdom, 2008.

[8] J. R. Kiniry and F. Fairmichael. Ensuring consistency between designs, documentation,formal specifications, and implementations. In *Proceedings of 12th International Symposium on Component-Based Software Engineering*, pages 242–261, 2009.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. The MIT Press, Cambridge, United States, 2001.

[10] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, Secaucus, United States, 1996.

[11] R. Alurand, T. A. Henzinger, and M. Y. Vardi. Theory in practice for system design and verification. *ACM SIGLOG News*, 2(1):46–51, 2015.

[12] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *Proceedings of the 2nd World Congress on Formal Methods*, pages 532–546, 2009.

[13] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[14] C. A. R. Hoare. Notes on an approach to category theory for computer scientists. *Constructive Methods in Computing Science*, 55:245–305, 1989.

[15] H. Kuang. *Towards a formal reactive autonomic systems framework using category theory*. PhD thesis, Concordia University, Montreal, Canada, 2013.[1]

[16] M. Zakeryfar. *Static analysis of a concurrent programming language by abstract interpreta- tion*. PhD thesis, Concordia University, Montreal, Canada, 2014.

[17] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue - Multiprocessors*, 3 (7):54–62, 2005.

[18] P. Grogono and B. Shearing. Concurrent software engineering: Preparing for paradigm shift. In *Proceedings of the First C* Conference on Computer Science and Software Engineering*, pages 99–108, 2008.

[19] J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11): 960–981, 1998.

[20] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[21] P. Grogono and B. Shearing. Modular concurrency: A new approach to manageable software. In *Proceedings of the Third International Conference on Software and Data Technologies*, pages 47–54, 2008.

1

[22] P. Grogono. The Erasmus project: Process oriented programming [online]. Available from: http://users.encs.concordia.ca/~grogono/Erasmus/erasmus.html [cited 14/7/16].

[23] Occam-π: blending the best of CSP and the pi-calculus [online]. Available from: http://www.cs.kent.ac.uk/projects/ofa/kroc/ [cited 14/7/16].

[24] P. Welch and N. Brown. Java communicating sequential processes [online]. Available from: http://www.cs.ukc.ac.uk/projects/ofa/jcsp/ [cited:14/7/16].

[25] P. Grogono and B. Shearing. A modular language for concurrent programming. Technical report, Concordia University, Montreal, Canada, 2006.

[26] P. Grogono and N. Jafroodi. A fair protocol for non-deterministic message passing. In *Pro- ceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 53–58, 2010.

[27] N. Jafroodi and P. Grogono. Implementing generalized alternative construct for Erasmus lan- guage. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 101–110, 2013.

[28] M. Zakeryfar and P. Grogono. Static analysis of concurrent programs by adapted vector clock. In *Proceedings of the Sixth International C* Conference on Computer Science and Software Engineering*, pages 58–66, 2013.

[29] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, pages 19–32, 1967.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, 1969. ISSN 0001-0782.

[31] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing, Boston, USA, 2006.

[32] A. Gosain and G. Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*, page 581.

[33] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Secaucus, United States, 1999.

[34] D. Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, pages 104–119, 2007.

[35] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[36] P. Cousot and R. Cousot. A galois connection calculus for abstract interpretation. In *Pro- ceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lan- guages*, pages 3–4, 2014.

[37] H. Herrlich and M. Husek. Galois connections categorically. *Journal of Pure and Applied Algebra*, 68(1-2):165–180, 1990.

[38] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the Association for Computing Machinery*, 19(3):137–147, 1976.

[39] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the Association for Computing Machinery*, 23(1):158–171, 1976.

[40] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194 – 206, 1973.

[41] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, 1989.

[42] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *SIGSOFT '94 Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 62 – 75, 1994.

[43] S. P. Masticola. *Static detection of deadlocks in polynomial time*. PhD thesis, Rutgers University, New Jersey, United States, 1993.

[44] D. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *TAV4 Proceedings of the symposium on Testing, analysis, and verification*, pages 21 – 35, 1991.

[45] A. Farzan and P. Madhusudan. Causal dataflow analysis for concurrent programs. In *Proceed- ings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 102 – 116, 2007.

[46] J. W. Voung. *Data flow analysis for concurrent programs using pata-race detection*. PhD thesis, University of California, San Diego, San Diego, United States, 2010.

[47] C. A. R. Hoare and J. Wickerson. Unifying models of data flow. In *Software and Systems Safety - Specification and Verification 2011*, pages 211–230, 2011.

[48] M. Barr and C. Wells. *Category theory for computing science*. Prentice-Hall, Upper Saddle River,United States, 2012.

[49] J. L. Fiadeiro. *Categories for software engineering*. Springer Berlin Heidelberg, Germany, 2005.

[50] S. Awodey. *Category theory*. The Clarendon Press, New York, 2006.

[51] G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford University Press, Oxford, United Kingdom, 1995.

[52] V. Sassone, M. Nielsen, and G. Winskel. Models for concurrency: towards a classification. *Theoretical Computer Science*, 170:297–348, 1996.

[53] T. T. Hildebrandt. *Categorical Models for Concurrency: Independence, Fairness and Dataflow*. PhD thesis, University of Aarhus, Aarhus, Denmark, 2000.

[54] E. Goubault and S. Mimram. Formal relationships between geometrical and classical models for concurrency. *CoRR*, abs/1004.2818, 2010.

[55] M. Zhu, P. Grogono, O. Ormandjieva, and P. Kamthan. Using category theory and data flow analysis for modeling and verifying properties of communications in the process-oriented lan- guage Erasmus. In *Proceedings of the Seventh C* Conference on Computer Science and Software Engineering*, pages 24:1–24:4, 2014.

[56] M. Zhu, P. Grogono, and O. Ormandjieva. Using category theory to verify implementation a- gainst design in concurrent systems. In *The 6th International Conference on Ambient Systems, Networks and Technologies*, pages 530–537, 2015.

[57] M. Zhu, P. Grogono, O. Ormandjieva, and H. Kuang. Using category theory to verify im- plementation against design in concurrent systems. In *The 7th International Conference on Ambient Systems, Networks and Technologies*, pages 700–704, 2016.

[58] M. Zhu, P. Grogono, O. Ormandjieva, and K. Zhao. Verifying consistency of process commu- nications between design and implementation of concurrent systems. In *Proceedings of the 2nd International Conference on Computer and Information Science and Technology*, pages 132–1–132–7, 2016.

[59] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.